Parallelizing Inductive Logic Programming in ASP

Final Report

Nikolay Paleshnikov

supervised by Alessandra Russo and Mark Law

June 24, 2016

Abstract

Although ASPAL manages to tackle any inductive learning task by transforming it into an abductive meta task and obtaining the solutions of the meta task from an answer set solver such as clingo, it performs poorly when faced with an intricate mode bias and a large Herbrand domain. On the one hand, the mode bias imposes restrictions on the shape and length of the skeleton rules constructed by ASPAL, i.e. on the hypothesis space of the task. On the other hand, the atoms of the skeleton rules are only partially instantiated in the meta task and their variables may be substituted with any of the ground terms from the Herbrand domain.

One of the preliminary steps in the solution of any answer set program is its grounding. It is exactly the grounding of the meta task that substantially slows down ASPAL or even renders it incapable of finding a solution. The possibility of a split of the original inductive learning task into subtasks has been recognized as a way to reduce the grounding size of the meta task and thus improve the scalability of ASPAL.

In this report, we propose a novel approach at parallelizing ASPAL, based on the notion of a dependency graph for the predicates of the given task. We prove its soundness for both optimal and general solutions, as well as its completeness in the case of general solutions. We subsequently present an algorithm computing the solutions of a given ASPAL task following our proposed approach, as well as its implementation in Python. An evaluation of the performance of our implementation shows its superiority to all known previous attempts at parallelizing ASPAL, as well as to ASPAL itself.

Acknowledgements

Above all, I would like to thank my supervisors Alessandra Russo and Mark Law for the inspiration they gave me for their research field and their undiminishing willingness to help me whenever I was struggling with my project. I am also grateful to my family for their unshakeable belief in their only son and their encouragement in times of trouble, as well as to my friends for sharing the most cheerful moments of our independent lives. I am greatly indebted to the German National Merit Foundation (die Studienstiftung des deutschen Volkes) for the financial and ideational support of my studies in both Germany and abroad. Lastly, I would like to thank Dirk Thissen, the Erasmus exchange coordinator at the technical university of Aachen, for giving me the chance to broaden my expertise in computing science, as well as my worldview during the time spent at Imperial College London.

Contents

1 Introduction

Inductive Logic Programming (ILP) has drawn huge interest towards itself for its expressive power and problem solving potential. Many efforts have been recently put into an attempt to reduce its computational complexity. One of the novel approaches is the ILP system ASPAL, encoding a brave ILP problem into an Answer Set Programming (ASP) task, thus transforming brave induction into an abductive search.

ASP has already proven itself as a chief paradigm in non-monotonic reasoning, giving appropriate semantics to normal logic programs. Moreover, many efficient ASP solvers have been developed such as *clingo*, which additionally provides an in-built grounding. One of the biggest limitations, however, is due to the typically large hypothesis space of the ILP task resulting in a huge grounding of the corresponding ASP meta task.

The primary aim of this project is to find a suitable decomposition of a given ILP task, formulated in the input language of ASPAL, whose components can then be solved in parallel and combined to obtain a solution of the original problem. The benefits from this approach are two-fold. First, the grounding of each sub-task is much smaller in size than the one of the whole task, thus making our algorithm more scalable. Secondly, the parallel execution is expected to bring an improved overall runtime while remaining resource-efficient.

In order to reduce the hypothesis space of an ILP task, the general form of any hypothesis in the solution set is predefined as part of the task. The head mode declarations specify which predicates may appear as a head of a rule in any of these hypotheses. When splitting a given ILP task, a key point is to set its head mode declarations apart into different subtasks. An important distinction is made between connected head declarations, whose predicates appear in the background knowledge or are allowed to be used as body atoms in the rules of any hypothesis, and disconnected head declarations.

Previous attempts at parallelizing ASPAL have succeeded in a limited scope and most notably produced PASPAL, which is able to handle ILP tasks, whose head declarations are all disconnected. It splits the ILP task into subtasks with one head declaration each, lets ASPAL solve them in parallel and outputs the product of their solutions as the solution of the original task. In case some of the head declarations are connected, however, PASPAL passes the original ILP task unaltered to ASPAL. This includes especially the cases of non-observational predicate learning. Some theoretical results have been proven to support the implementation.

There has also been a proposal for a solution in the general case, when connected head predicates are also present. Due to its faulty definition and out of our recognition for the greater potential of a splitting procedure based on graph theory, we have developed our own theoretical framework, which provides us with a dependency graph for the predicates of a given ILP task. Each of the edges of the dependency graph represents a binary relation between predicates called a dependency as opposed to the less expressive unary relation of connectedness described above. We have been able to prove the correctness and completeness of this representation and thereby the suitability of our theoretical framework to capture dependencies between the predicates of an ILP task.

Upon construction of the dependency graph, we have taken account of circular dependencies and non-observational components to produce a reduced dependency graph, whose connected components correspond to pairwise disconnected sets of predicates. We were able to formulate a split of any ILP task into subtasks for each of these sets, which are solved by ASPAL in parallel. The product of their solutions is proven to coincide with the solution set of the original task in respect to both general and optimal solutions.

Once we have split the task into subtasks for each of the pairwise disconnected sets of predicates, we have looked into possibilities for splitting further within each of these sets. We have once again turned to the reduced dependency graph and devised a solution computation procedure for each of its connected components, whose subtasks correspond to single nodes of the graph. While we have proven that it is correct but not complete in optimal solution settings, we have also proven it is correct and complete when general solutions are considered.

Lastly, we have formulated an alternative approach to parallelizing ILP tasks, which splits the original task into subtasks, each with a single head declaration and a set of abducibles for all the head declarations. Due to time constraints on this project, the proof and implementation of this approach are left to future research in the area. We have also pointed out the possibility of integrating the split based on the connected components of the dependency graph with a solution computing procedure using abducibles to split further within each of them. Further possible improvements include an iterative computation mode with either increasingly suboptimal solutions or theory revision on incomplete hypotheses using RASPAL.

1.1 Summary of contributions

This project has made the following contributions:

- Development of a theoretical framework capturing dependencies between predicates of a given ILP task. It has provided us with a dependency graph of the ILP task and its reduction, on the basis of which we split the original task into subtasks

- Definition of a split for an arbitrary ILP task into subtasks for pairwise disconnected sets of predicates. Any of these sets comprises the predicates of a connected component of the reduced dependency graph. The subtasks are solved by ASPAL in parallel and the product of their solutions is proven to provide us with the solution of the original ILP task when either general or optimal solutions are considered.

- Inductive definition of a split for the predicates of a connected component of the reduced dependency graph. While predicates at different depths of the graph are taken care of sequentially, all of the subtasks for predicates at the same depth are solved by ASPAL in parallel. The solution computation procedure based on this definition is proven to be sound but not complete in terms of optimal solutions, whereas it is also proven to be both sound and complete in terms of general solutions.

- Construction of an algorithm encompassing both splits described above and its implementation in Python. In the tests we have performed, it has always been able to produce a subset of (slightly sub-) optimal solutions under optimal solution settings and all of the general solutions under general solution settings. Given an ILP task with sufficiently big mode bias and observations, it substantially improves the runtime of ASPAL. It also potentially prevents ASPAL from ‘out of memory’ crashes due to the reduced grounding size of each of the subtasks compared to the original task.

- Assessment of the performance of our python script in comparison with other existing approaches at parallelizing ILP and ASPAL itself.

1.2 Report structure

The background section deals mostly with logic definitions, which are used throughout the whole report. Their succinct explanations and some examples are provided in subsections 2.1 – 2.6. In the following subsections 2.7 – 2.8, efficient approaches are formulated to problems leading up to an ILP task, whose specification and solution steps are handled in section 2.9. Some useful notions from graph theory, which we refer to in the process of construction and analysis of the dependency graph of a given inductive learning task, are touched upon in 2.10. Once the background of the research field is familiar to the reader, our own contributions to the parallelization of ILP are expounded in section 3. After juxtaposing them to previous milestones in the parallelization of ASPAL in section 6, we provide some practical evidence for their efficiency in section 7. In the last section 8, we summarize the importance and value of what we have done and suggest directions for further research.

2 Background

A basic knowledge of first-order logic is presupposed. In case the reader is not familiar with it, it is advisable to read some of the ubiquitous literature on this topic, e.g. [1]. A good overview of the theory and methods of ILP is to be found in [2].

According to the notation in Prolog, variables begin with an upper-case letter, whereas constants – with a lower-case letter. iff stands for 'if and only if' to mark the logical equivalence of two statements. $\perp$ denotes the logical truth value *false*.

2.1 Logic Programs and Normal Logic Programs

Definition 2.1.1. The alphabet of logic programs we are going to use throughout this report is defined over the universe U and consists of the following elements:

- variables X, Y, Z, denoting any element $e \in U$
- constants a, b, c, or zero-arity function symbols, denoting a fixed element $e \in U$
- function symbols $f, g, h : U^n \to U$ with a fixed arity n
- propositional letters p, q, or zero-arity predicate symbols
- predicate symbols $p, q \subseteq U^n$ with a fixed arity n

All the rules containing variables are implicitly universally quantified. Therefore, rules containing variables are considered only as an abbreviation of the set of rules, where each variable is instantiated with any element of the universe.

A term is a constant, a variable or a function applied to terms according to its arity. An atom is either a proposition or is constructed by applying any of the predicate symbols to terms corresponding in number to its arity.

Definition 2.1.2. For an atom $a = p(t_1, ..., t_n)$, we define $pred(a) = p$. For a set of atoms s, we denote $\{pred(a) \mid a \in s\}$ as $pred(s)$.

The logic programs we consider in this report are all in conjunctive normal form, i.e. constructed as a conjunction of disjunctions: $\bigwedge \bigvee l_i$, where each disjunction $\bigvee l_i$ is called a clause or a rule and each literal l_i is either positive, i.e. an atom, or negative, i.e. the negation of an atom. There are two types of negation – the classical negation $\neg$ and Negation by failure (NBF) **not**. NBF derives its semantics from the 'closed world assumption', stating that everything to be known is part of the logic program.

Definition 2.1.3. **not** p evaluates to true iff p cannot be derived.

Remark 2.1.4. We use '$\sim$' to denote the opposite literal according to NBF, i.e. $\sim p =$ **not** p and $\sim$ **not** $p = p$.

A normal rule contains only atoms, each of which is either positive or NBF. Normal logic programs consist solely of normal rules. They are usually interpreted as a set due to the commutative property of the logical operator $\wedge$ and written in an arbitrary order of their rules, separated from each other by a dot. Each rule r is commonly written as an implication of the form

$a \leftarrow b_1, ..., b_n,$ **not** $c_1, ...,$ **not** c_m.

where $n, m \geq 0$, a is the head of the rule, i.e. $head(r) = a$, whereas the positive and negative atoms in the body are denoted as the sets $body^+(r) = \{b_1, ..., b_n\}$ and $body^-(r) = \{c_1, ..., c_m\}$. Finally, $body(r) = body^+(r) \cup body^-(r)$ and $atoms(r) = head(r) \cup body(r)$.

Remark 2.1.5. In normal logic programs, the head of a rule may only be empty or a single positive atom. A headless rule is called a constraint and its head is interpreted as $\bot$, thus filtering out possible solution candidates, in which the body of the constraint evaluates to true. A bodiless rule is universally true and is therefore called a fact. We call any other rule an ordinary rule.

Normal logic programs not containing any NBF, i.e. whose rules all have $m = 0$, are called definite logic programs.

In the rest of this report, normal logic programs are typically referred to simply as logic programs. Whenever a different kind of a logic program is considered, this will be explicitly mentioned.

Logic programs have become a standard for logical computations and encompass the valid input of various logic programming languages such as Prolog. They also form the basis for the input of ASP solvers like clingo.

2.2 Minimal Herbrand models

It is a central issue to evaluate normal logic programs. In order to do so algorithmically, they need to be normalized further.

A ground term is either a constant or is obtained by a finite amount of function applications at constants. The set of all ground terms of a logic program P is denoted as its Herbrand domain $HD(P)$.

Example 2.2.1. Let P denote the logic program

$p(a)$.
$q(b)$.
$r(f(X), g(Y)) \leftarrow p(X), q(Y)$.

$\{a, b, f(a), g(b), f(f(a)), f(g(b))\}$ is, among many others, a subset of $HD(P)$.

The Herbrand domain of a finite logic program is infinite iff it contains at least one function symbol with a non-zero arity and at least one constant. This is clearly the case with P from example 2.2.1.

A ground logic program is one that does not contain any variables. To 'ground' a logic program P or a rule r means to consistently substitute each of its variables with a ground term.

A safe rule is one in which for each occurring variable, there is at least one positive atom in the body containing that variable.

There are various ways to restrict the grounding of a safe rule to a (usually finite) subset with the same semantics as the whole grounding, e.g. by taking into account only the relevant ground terms as opposed to the whole Herbrand domain. In this report, we write $ground(P)$ for the relevant grounding of a logic program P.

Example 2.2.2. As an ongoing example, consider the logic program Q:

$p(X) \leftarrow$ **not** $q(X), r(X).$
$q(X) \leftarrow r(X), s.$
$s \leftarrow$ **not** $t.$
$\leftarrow s, t.$
$r(a).$

Its relevant grounding is

$ground(Q) =$
$p(a) \leftarrow$ **not** $q(a), r(a).$
$q(a) \leftarrow r(a), s.$
$s \leftarrow$ **not** $t.$
$\leftarrow s, t.$
$r(a).$

A ground atom is either a proposition or is obtained by applying any of the non-zero arity predicate symbols to ground terms. The set of all ground atoms of a logic program P is denoted as its Herbrand base $HB(P)$.

Example 2.2.3. For Q from example 2.2.2, we have $HD(Q) = \{a\}$ and $HB(Q) = \{p(a), q(a), r(a), s, t\}$.

Definition 2.2.4. A Herbrand interpretation $\mathfrak{I}$ of a logic program P is a subset of $HB(P)$ that assigns a truth value to all atoms in $HB(P)$. Each atom in $\mathfrak{I}$ evaluates to true and the residual ones — to false, i.e. $a \in \mathfrak{I} \iff \mathfrak{I} \models a$.

In a supported interpretation $\mathfrak{I}$, each atom $a \in \mathfrak{I}$ is a head of a rule whose body is true in $\mathfrak{I}$.

Definition 2.2.5. Let P be a logic program. A Herbrand model M of P is a Herbrand interpretation satisfying each clause of P, i.e.

- each fact is a member of M
- whenever M satisfies the body of an ordinary clause, it satisfies its head as well, i.e. $\forall r \in P\ (head(r) \neq \bot \wedge body^+(r) \subset M \wedge body^-(r) \cap M = \emptyset) \Rightarrow head(r) \in M$
- M does not satisfy the body of any constraint

A minimal Herbrand model of P is one that does not have a proper subset, which is also a model of P. A unique minimal Herbrand model is also called a least Herbrand model.

Lemma 2.2.6. Each definite logic program P has a least Herbrand model.

It is the intersection of all Herbrand models of P and can be constructed as the least fixed point of the immediate consequence operator T_P of P. For details, see [3].

Example 2.2.7. Q from example 2.2.2 is satisfiable and has two models:

$\{q(a), r(a), s\}$ and $\{p(a), r(a), t\}$,

of which only $\{q(a), r(a), s\}$ is supported, because $r(a)$ is a fact, t is not the head of any rule in Q, so that t cannot be derived and **not** t evaluates to true, thus yielding s. $q(a)$ is then derived from $r(a)$ and s.

They are both minimal as no proper subset of theirs is a model of Q.

$\{p(a), r(a), s, t\}$ is not a model, because $\{p(a), r(a), s, t\} \not\models \ \leftarrow s, t.$ and therefore $\{p(a), r(a), s, t\} \not\models Q$.

Herbrand models don't usually give us the intended semantics of a normal logic program containing NBF. Therefore, the stable model semantics have been introduced.

2.3 Stable Models

All the definitions in this section are according to [4]. To define answer sets, we need the notion of a reduct, providing a partial evaluation of a logic program with respect to NBF.

Definition 2.3.1. Let P be a ground logic program and X a set of atoms. To obtain the reduct P^X, one must delete any rule r in P with $\mathit{body}^-(r) \cap X \neq \emptyset$ and every negative literal in the bodies of the remaining rules.

The reduct P^X does not contain any NBF, is also by definition a definite logic program and has a least Herbrand model according to lemma 2.2.6.

Definition 2.3.2. The set X is a stable model of the normal logic program P iff X is the least Herbrand model of P^X.

Remark 2.3.3. In the context of normal logic programs, the terms 'stable model' and 'answer set' are used interchangeably.

A stable model is minimal and supported in compliance with its definition and lemma 2.2.6. The reverse does not hold in general, as apparent from the following example. For this reason, there is no other way to compute the answer sets of a given program than to build its reducts according to each answer set candidate.

Example 2.3.4. Find the stable models of the following ground program P:

$a \leftarrow a.$
$b \leftarrow$ **not** $a.$

In order to do so, build the reduct of P with respect to any subset of the power set of HB(P), i.e. $\mathbb{P}(\{a, b\})$:

$P^{\emptyset} = \{a \leftarrow a.\ b.\}$ with minimal Herbrand model $\{b\} \neq \{\emptyset\}$
$P^{\{a\}} = \{a \leftarrow a.\}$ with minimal Herbrand model $\{\emptyset\} \neq \{a\}$
$P^{\{b\}} = \{a \leftarrow a.\ b.\}$ with minimal Herbrand model $\{b\}$
$P^{\{a,b\}} = \{a \leftarrow a.\}$ with minimal Herbrand model $\{\emptyset\} \neq \{a, b\}$

Therefore, only $\{b\}$ is a stable model of P. Note that $\{a\}$ is also minimal as no proper subset of $\{a\}$ entails all the clauses of P. It is supported as well, since a supports itself via the clause $a \leftarrow a$. It follows that it is necessary but not sufficient for a stable model to be supported and minimal.

Example 2.3.5. Show whether the minimal Herbrand models of Q from example 2.2.2 are stable:

$ground(Q)^{\{q(a),r(a),s\}} =$

$q(a) \leftarrow r(a), s.$
$s.$
$\leftarrow s, t.$
$r(a).$

and its minimal Herbrand model is $\{q(a), r(a), s\}$, therefore it is also stable.

$ground(Q)^{\{p(a),r(a),t\}} =$

$p(a) \leftarrow r(a).$
$q(a) \leftarrow r(a), s.$
$\leftarrow s, t.$
$r(a).$

and its minimal Herbrand model is $\{p(a), r(a)\} \neq \{p(a), r(a), t\}$, therefore it is not stable.

Arising from the fact that a single logic program can have several answer sets, we must define the notion of entailment in a non-customary way.

Definition 2.3.6. A logic program P bravely entails a ground atom a, i.e. $P \models_b a$, if it is a member of at least one answer set of P. P cautiously entails a ground atom a, i.e. $P \models_c a$, if it is a member of all answer sets of P.

2.4 Extensions of the Answer Set semantics

There are some useful extensions of the answer sets semantics, two of which we are going to need later on in this report – count aggregates and optimization statements. The syntax we use is compatible with the input language of clingo version 3, described in further detail in [5]. This is also the version used by ASPAL.

Definition 2.4.1. The count aggregate has the following syntax:

$lower\ \{l_1, ..., l_n\}\ upper$,

and evaluates to true iff the number T of true literals in the set $\{l_1, ..., l_n\}$ satisfies $lower \leq T \leq upper$, thus expressing a cardinality constraint.

Definition 2.4.2. An optimization statement is a weak constraint giving priority to answer sets with desired properties (for further details on weak constraints, consult [5]). An optimization statement of the form

$\#minimize[a_1 = w_1, ..., a_n = w_n].$

lets clingo compute an answer set as minimizing the weighted sum of atoms in the set $\{a_1, ..., , a_n\}$, i.e. one with the lowest sum $\sum\limits_{i \in \{1...n\ |\ a_i \in as\}} w_i$.

Example 2.4.3. Let clingo evaluate the following program:

```
1 {a,b,c} 2.
#minimize[a=2,b=1,c=1].
```

First, it determines that there are six answer sets: {a},{b},{c},{a,b},{a,c} and {b,c}. Then, it computes their weights to be 2,1,1,3,3 and 2, respectfully. Finally, it outputs either of {b} and {c} as an optimal solution.

2.5 Stratification

The idea of stratification is to divide the logic program into several parts ('strata'), which are not mutually dependent on each other and can thus be evaluated sequentially. This way, the search for a suitable interpretation is broken into smaller parts, easier to be solved.

Definition 2.5.1. A normal logic program P is stratified iff it can be partitioned into disjoint subprograms $P_1 \cup ... \cup P_n = P$ so that for each predicate p occurring in P, all clauses with p in their head are contained in exactly one subprogram $P_i, 1 \leq i \leq n$. Moreover, for each $i \in \{1, ..., n\}$ and rules $r_{ij} \in P_i$, altogether m_i in number, the definitions of the positively occurring atoms are contained within the preceding strata including the one in question, i.e.
$\forall i \; \bigcup_{j=1}^{m_i} \{r | a \in \mathit{body}^+(r_{ij}) \wedge r \in P \wedge a = \mathit{head}(r)\} \subset \bigcup_{k=1}^{i} P_k$, whereas the definitions of the negatively occurring atoms are contained within the preceding strata excluding the one in question, i.e.
$\forall i \; \bigcup_{j=1}^{m_i} \{r | a \in \mathit{body}^-(r_{ij}) \wedge r \in P \wedge a = \mathit{head}(r)\} \subset \bigcup_{k=1}^{i-1} P_k$.

Example 2.5.2. $\mathit{ground}(Q)$ from example 2.2.2 is stratified and could be partitioned for instance in the following way:

$Q_1 = \{q(a) \leftarrow r(a), s.\ s \leftarrow \ \textbf{not}\ t.\ \leftarrow s, t.\ r(a).\}$
$Q_2 = \{p(a) \leftarrow \ \textbf{not}\ q(a), r(a).\}$

Then each of the minimal Herbrand models of Q is the union of a minimal Herbrand model M_1 of Q_1 and M_2 of $Q_2 \cup M_1$:

1. $M_1 = \{q(a), r(a), s\}$, $M_2 = \emptyset$ and $\{q(a), r(a), s\} = M_1 \cup M_2$
2. $M_1 = \{r(a), t\}$, $M_2 = \{p(a)\}$ and $\{p(a), r(a), t\} = M_1 \cup M_2$

If we were to add the rule $t \leftarrow \ \textbf{not}\ s.$ to $\mathit{ground}(Q)$, the resulting program Q' would not be stratified because of the 'recursion through negation' in rules:

$s \leftarrow \ \textbf{not}\ t.$
$t \leftarrow \ \textbf{not}\ s.$

2.6 Splitting Set Theorem

A generalization of stratification that sometimes even works with non-stratified programs is the splitting set theorem, dealing with answer sets. The splitting set theorem provides us with a guarantee that after an appropriate break-up of a logic program, the union of the answer sets of its subprograms matches the answer set of the original program.

Definition 2.6.1. A splitting set U of a logic program P is a subset of HB(P), each atom of which is defined solely by atoms in the subset, i.e. U has the following property:

$\forall r \in P\ head(r) \in U \rightarrow atoms(r) \subset U$.

U defines the splitting of P into a bottom ($bot_U(P)$) and a top part ($top_U(P)$) with $bot_U(P) = \{r | a \in U, r \in P, a = head(r))\}$, $top_U(P) = P \setminus bot_U(P)$

The empty set and $HB(P)$ are trivially splitting sets of P, though of no practical value.

Since all atoms of a splitting set U of P are defined within U, we can find an answer set of $bot_U(P)$ without considering $top_U(P)$. For the computation of an answer set of $top_U(P)$, however, we need to take a preliminary step to partially evaluate the rules in $top_U(P)$ with respect to a given answer set X of $bot_U(P)$. We thus start from X and use it as interpretation over U to assign truth values to the atoms in U occurring in the bodies of rules in $top_U(P)$.

Definition 2.6.2. A rule r is in the partially evaluated top part of P according to a splitting set U and some answer set X of $bot_U(P)$, denoted as $e_U(top_U(P), X)$, iff there is a rule q in P with $body^+(q) \cap U \subseteq X$ and $body^-(q) \cap X = \emptyset$, so that $head(p) = head(q)$, $body^+(p) = body^+(q) \setminus U$ and $body^-(p) = body^-(q) \setminus U$.

Theorem 2.6.3. (*splitting set theorem*) Let P be a logic program and U a splitting set of P. S is an answer set of P iff there are X and Y so that $S = X \cup Y$, $X \cap Y = \emptyset$, X is an answer set of $bot_U(P)$ and Y – an answer set of $e_U(top_U(P), X)$.

A proof of this theorem for the more general case of disjunctive logic programs can be found in [4].

Example 2.6.4. Let us apply the splitting set theorem at $ground(Q)$ from example 2.2.2 with splitting set $U = \{q(a), r(a), s, t\}$:

$p(a) \leftarrow$ **not** $q(a), r(a).$

$q(a) \leftarrow r(a), s.$
$s \leftarrow$ **not** $t.$
$\leftarrow s, t.$
$r(a).$

To acquire the only answer set $\{q(a), r(a), s\}$ of $ground(Q)$, ascertain that $X = \{q(a), r(a), s\}$ is an answer set of $bot_U(ground(Q))$ and $\emptyset$ an answer set of $e_U(top_U(ground(Q)), X)$.

Now consider Q' from example 2.5.2. Its split according to U is

$p(a) \leftarrow$ **not** $q(a), r(a).$

$q(a) \leftarrow r(a), s.$
$s \leftarrow$ **not** $t.$
$t \leftarrow$ **not** $s.$
$\leftarrow s, t.$
$r(a).$

The first answer set $\{q(a), r(a), s\}$ of Q' is obtained in the same fashion as above, the second one is $\{p(a), r(a), t\} = \{r(a), t\} \cup \{p(a)\}$ with $\{r(a), t\}$ – an answer set of $bot_U(Q)$ and $\{p(a)\}$ – an answer set of $e_U(top_U(P), X)$.

2.7 Logical reasoning techniques

In the construction of Herbrand models, reducts and answer sets, we have only used logical reasoning to infer something from the facts, as specified in remark 2.1.5, or from the bottom set of a splitting set, as defined in 2.6.1, based on the rules of our program. While this technique, called deduction, has played the most important role so far, its counterparts, known as in- and abduction, are going to aid us substantially throughout the rest of this report. To clearly distinguish between the three of them, it is helpful to first model logical inference and then identify the unknown element we are trying to construct applying each of them so that the inference holds at least in our specific case.

Logical inference typically has the following form:

premisses $\xrightarrow{\text{a set of rules}}$ conclusion, e.g.

$\textcircled{1}$ It is cloudy. $\xrightarrow{\textcircled{2}\text{Whenever it is cloudy, it rains.}}$ $\textcircled{3}$ It rains.

With deduction, we start from $\textcircled{1}$ and $\textcircled{2}$ to derive $\textcircled{3}$. No additional knowledge is gained as the conclusion is necessarily true.

Abduction helps us find $\textcircled{1}$ such that we can infer $\textcircled{3}$ with the rule $\textcircled{2}$. It identifies a feasible cause, from which the observed effect follows according to the given rule. In our problem domain, there usually is an infinite set of possible premisses, therefore it gets restricted by a specification of *abducibles*, i.e. a set of premisses to pick from.

In the case of induction, we search for $\textcircled{2}$ given $\textcircled{1}$ and $\textcircled{3}$, i.e. we try to construct a general rule explaining the observed effect given its cause. To reduce the amount of exploration to rules of a certain form, mode declarations are used, as described in section 2.9.

Remark 2.7.1. Note that only the results drawn by deduction hold in general. Induction tends to overgeneralize the desired rules in the absence of negative observations, while abduction may produce too many answer candidates if the rules are not restrictive enough.

2.8 Abduction in ASP

The definitions in this and the following section are according to [6] and [7].

Definition 2.8.1. An abductive logic programming task has the form $< KB, Ab, IC >$, where the background knowledge or knowledge base (KB) is a normal logic program, $Ab \subset HB(KB)$ represents the set of abducibles and IC is a set of integrity constraints. To solve the abductive task against a given set of observations Ob, consisting of positive (E^+) and negative examples (E^-), i.e. subsets of $HB(KB)$, we need to find $\Delta \in ALP(< KB, Ab, IC >, Ob)$ with $\Delta \subseteq Ab$ so that our solution, combined with the background knowledge, explains the observations and remains consistent with the background knowledge and the integrity constraints. Formally, the following three conditions must hold:

- $\Delta \subset Ab$
- $\Delta \cup KB \models Ob$
- $\Delta \cup KB \cup IC$ is consistent, i.e. $\Delta \cup KB \cup IC \not\models \bot$

In order to solve an abductive logic programming task, we must transform the tuple $(< KB, Ab, IC >, Ob)$ into a logic program P, written in the input language of clingo. Assume that KB and IC are already of the desired kind and $Ab = \{a_1, ..., a_n\}$. We construct the aggregate $0\ \{a_1; ...; a_n\}\ n$, whose upper bound n equals the cardinality of the set of abducibles $|Ab|$, as well as the goal clause $g \leftarrow \bigwedge(E^+ \cup \{\ \textbf{not}\ a \mid a \in E^-\})$ and its corresponding constraint $\leftarrow\ \textbf{not}\ g$. We subsequently add them to $KB \cup IC$ to obtain a logic program P and let clingo find its answer sets $AS(P)$. The abductive solutions are then the intersections of these answer sets with the set of abducibles, i.e. $\{as \cap Ab\ |as \in AS(P)\}$.

Example 2.8.2. Solve the task $(< KB, Ab, IC >, Ob)$ with
KB = { girl(mary). girl(sarah). receives_flowers(X) :- young_man(Y), likes(Y,X). young_man(X) :- girl(Y), likes(X,Y). }
Ab = {likes(albert,mary),likes(albert,sarah),likes(john,mary),likes(john,sarah)},
IC = { :- girl(X), young_man(Y), young_man(Z), Y != Z, likes(Y,X), likes(Z,X). } and Ob = {receives_flowers(mary), receives_flowers(sarah)}.
The program to be solved by clingo is then the following:

```
girl(mary).
girl(sarah).
receives_flowers(X) :- young_man(Y), likes(Y,X).
young_man(X) :- girl(Y), likes(X,Y).
:- girl(X), young_man(Y), young_man(Z), Y != Z,
    likes(Y,X), likes(Z,X).
0{likes(albert,mary),likes(albert,sarah),
   likes(john,mary),likes(john,sarah)}4.
goal :- receives_flowers(mary), receives_flowers(sarah).
:- not goal.
```

The solutions of the abductive task are then

{likes(john,mary),likes(albert,sarah)}, {likes(john,mary) likes(john,sarah)}, {likes(albert,mary), likes(john,sarah)} and {likes(albert,mary), likes(albert,sarah)}.

We could have added the optimization statement

```
#minimize[likes(albert,mary)=3, likes(john,mary)=2,
          likes(john,sarah)=1].
```

to find a solution conforming to our belief that it is less likely for John to like Mary than – Sarah, but it is even more unlikely for Albert to like Mary. In that case, there is only one optimal solution – {likes(john,mary),likes(albert,sarah)}.

2.9 Induction in ASP

To solve an inductive learning task with background knowledge B, positive examples E^+ and negative examples E^-, we need to construct a set or rules H, called a hypothesis, which is consistent with the background knowledge and together with it explains the examples, i.e. $H \cup B \nvDash \bot \land \forall e \in E^+ \ (H \cup B \models e) \land \forall e \in E^- \ (H \cup B \nvDash e)$.

One important issue arises from the answer set semantics of our solution. There are two different forms of an inductive task, corresponding to the two distinct kinds of entailment as described in definition 2.3.6.

Definition 2.9.1. Let B be the background knowledge, E^+ and E^- the positive and negative examples and AS – the set of answer sets of $B \cup H_c$. A cautious inductive solution H_c is one such that $B \cup H_c$ has at least one answer set, cautiously entails all positive examples and does not bravely entail any of the negative examples, i.e.
$AS \neq \emptyset \land \forall e \in E^+ \ (\forall as \in AS \ e \in as) \land \forall e \in E^- \ (\forall as \in AS \ e \notin as)$.

Definition 2.9.2. Let B, E^+ and E^- be as above and AS denote the set of answer sets of $B \cup H_b$. A brave inductive solution H_b is one that has an answer set satisfying all positive examples and not satisfying any of the negative examples, i.e. $\exists as \in AS \ s.t. \ \forall e \in E^+ \ (e \in as) \land \forall e \in E^- \ (e \notin as)$.

From now on, we assume all inductive tasks are defined under the semantics of brave entailment.

To restrict the hypothesis space, mode declarations are introduced. A distinction is made between variables, which are expected to already be instantiated at the time of evaluation of the respective atom, i.e. variables appearing in the head or in a preceding body atom (input variables), and variables to be instantiated as an outcome of the evaluation (output variables). The mode declarations M are accordingly defined over a (possibly NBF) predicate symbol p and ground literals $g_1, ..., g_n$, containing type constants of three different kinds, referring to either input variables (*+type*), output variables (*-type*) or ground terms (*#type*). There are head declarations – M_h, of the form $modeh(p(g_1, ..., g_n))$, and body declarations – M_b, of the form $modeb(p(g_1, ..., g_n))$ or $modeb(\ \textbf{not}\ p(g_1, ..., g_n))$, which specify the form of the head and body atoms of all permissible rules.

We extend definition 2.1.2 of the *pred* operator over the background knowledge and the mode declarations of an inductive learning task.

Definition 2.9.3. For an inductive learning task $< B, E, M >$, we write $pred(B)$ for $\{pred(a) \mid a \in HB(B)\}$. For $h = modeh(p(t_1, ..., t_n)) \in M_h$, we define $pred(h) = p$. Similarly, for $b = modeb(p(t_1, ..., t_n)) \in M_b$, we define $pred(b) = p$. Finally, we write $pred(M)$ for $\{pred(m) \mid m \in M_h \cup M_b\}$.

We will need the notion of a predefined observed predicate later in the report.

Definition 2.9.4. A predefined observed predicate is a predicate that appears in B and E, but not in M_h.

An atom is compatible with a mode declaration *dec* iff it is built by application of *pred*(*dec*) according to its arity and type constants. A compatible rule is one that consists of compatible atoms and therefore matches a subset of the mode declarations. To obtain the general structure of all compatible rules, the skeleton rules S_M are constructed (the following definition is a simplified version of an analogical one in [8]).

Definition 2.9.5. Given a set of mode declarations M, a skeleton rule $r \in S_M$ is a normal rule r of the form $h \leftarrow b_1, ..., b_n, t_1, ..., t_m$ such that:

- h is an atom constructed from one of the head mode declarations by replacing all the input variables and constants with different variables
- Each b_i is a literal constructed from one of the body mode declarations by replacing:
  - all the input variables with variables that have already occurred previously as either the output variable of another body atom or as an input variable in the head
  - all the constants and output variables with new variables
- each variable that has not replaced a constant has an extra type atom t_i added to the body, so that we can ensure that each variable takes exactly one type
- (*number restrictions*) the number of literals in the body, the total number of variables that have not replaced a constant and the recall of each predicate in a single rule are bounded by given constants

Later on in this report, we write $typeBodyAtoms(r)$ for the set $\{t_1, ..., t_m\}$ of body atoms of a skeleton rule r used to enforce the types and
$nonTypeBodyAtoms(r) = body(r) \setminus typeBodyAtoms(r)$.

Example 2.9.6. For $M_h = \{modeh(p(+t)).\}$ and
$M_b = \{modeb(q(+t)).\ modeb(r(+t)).\}$, ASPAL constructs the following set of skeleton rules:
$S_M = \{$
$p(A) \leftarrow t(A).$
$p(A) \leftarrow t(A), r(A).$
$p(A) \leftarrow t(A), q(A).$
$p(A) \leftarrow t(A), r(A), q(A).\}$

Given the skeleton rules S_M, the inductive learning task $< B, E, M >$ can be formulated as an abductive meta task $meta(< B, E, M >)$ with $B \cup aspal\_encoding(S_M) \cup example\_clauses$ as its background knowledge and a predicate representation called the meta encoding $meta(r)$ of each compatible rule $r \in S_M$ as its set of abducibles.

For each rule r in S_M, $meta(r)$ is appended as an additional body atom. These amended rules constitute $aspal\_encoding(S_M)$.

For $E^+ = \{e_1^+, ..., e_n^+\}$ and $E^- = \{e_1^-, ..., e_m^-\}$, *example_clauses* consists of:

$goal \leftarrow e_1^+, ..., e_n^+,\ \textbf{not}\ e_1^-, ...,\ \textbf{not}\ e_m^-.$
$\leftarrow\ \textbf{not}\ goal.$

For $S_M = \{r_1, ..., r_n\}$, the abducibles are translated into the following count aggregate for the clingo input file:

$0\{meta(r_1), ..., meta(r_n)\}n.$

ASPAL then uses clingo to solve $meta(< B, E, M >)$ and outputs as a solution the set of rules $\{r \mid r \in S_M \wedge meta(r) \in as\}$, whose meta encoding is abduced in a given answer set as of $meta(< B, E, M >)$. The interested reader is advised to read [7] for the underlying theory.

Example 2.9.7. For S_M from example 2.9.6,

$aspal\_encoding(S_M) = \{$
$p(A) \leftarrow t(A), meta(r_1).$
$p(A) \leftarrow t(A), r(A), meta(r_2).$
$p(A) \leftarrow t(A), q(A), meta(r_3).$
$p(A) \leftarrow t(A), r(A), q(A), meta(r_4).\}$

and the count aggregate for the abducibles is

$0\{meta(r_1), meta(r_2), meta(r_3), meta(r_4)\}4.$

Definition 2.9.8. Let a set of clauses B denote the background knowledge, a pair of sets of ground facts $E =< E^+, E^- >$ – the positive and negative examples and a pair $M =< M_h, M_b >$ – the head and body mode declarations. $ASPAL(< B, E, M >)$ denotes the set of (general) solutions of the inductive learning task $< B, E, M >$. Each of these solutions consists of the rules whose meta encoding is abduced in any of the answer sets of the abductive learning task $meta(< B, E, M >)$.

Definition 2.9.9. In the settings of the previous definition,
$ASPAL^*(< B, E, M >) \subseteq ASPAL(< B, E, M >)$ denotes the set of optimal solutions of the inductive learning task $< B, E, M >$. Each of these solutions consists of the rules whose meta encoding is abduced in any of the optimal answer sets of the abductive learning task $meta(< B, E, M >)$.

For $S_M = \{r_1, ..., r_n\}$, this causes ASPAL to add the following minimization statement to the abductive meta task, weighting each rule as the number of its body atoms not used to enforce the types plus one (in order to account for the head atom as well):

$$\#minimize[meta(r_1) = \ |nonTypeBodyAtoms(r_1)| + 1\ , ..., \\ meta(r_n) = \ |nonTypeBodyAtoms(r_n)| + 1].$$

The optimal solutions ASPAL returns then correspond to the optimal solutions clingo outputs when called with the option `--opt-all`.

The only difference in the input language of ASPAL from the input language of clingo are the mode declarations and the way observations are specified. $example(p, 1).$ denotes a positive example of p, whereas $example(p, -1).$ – a negative one.

Example 2.9.10. Consider the inductive learning task $< B, E, M >$ with $B = \{r(a).\ t(a).\ t(b).\}$, $E^+ = \{p(a)\}$, $E^- = \{p(b)\}$ and M as specified in example 2.9.6. The ASPAL encoding of this task is

```
modeh(p(+t)).
modeb(q(+t)).
modeb(r(+t)).

r(a).
t(a).
t(b).

example(p(a),1).
example(p(b),-1).
```

whereas the abductive meta task can be formulated as the following clingo program:

```
% count aggregate for the abducibles
0{meta(r_1),meta(r_2),meta(r_3),meta(r_4)}4.

% background knowledge
r(a).
t(a).
t(b).

% ASPAL encoding of the skeleton rules
p(A) :- t(A), meta(r_1).
p(A) :- t(A), r(A), meta(r_2).
p(A) :- t(A), q(A), meta(r_3).
p(A) :- t(A), r(A), q(A), meta(r_4).
% example clauses
goal :- p(a), not p(b).
:- not goal.
```

So as to compute just the optimal solutions, ASPAL adds the following optimization statement to the program from above:

```
#minimize[meta(r_1)=1,meta(r_2)=2,meta(r_3)=2,meta(r_4)=3].
```

The only optimal solution clingo computes is the answer set $\{r(a), t(a), t(b), meta(r_2), p(a), goal\}$, which corresponds to the only optimal solution of the inductive task

```
p(A) :- t(A), r(A).
```

More elaborate examples of programs, written in the input language of ASPAL, are given in the following sections.

As we opt for parallelizing a given ASPAL task, we are going to split it and then solve each of its subtasks in parallel. So as to be able to combine two ASPAL solution sets afterwards, we introduce the notion of a product of two sets of sets.

Definition 2.9.11. In this report, we redefine the Cartesian product operator $\times$ for sets of sets S_1 and S_2 as $S_1 \times S_2 = \{s_1 \cup s_2 \mid s_1 \in S_1 \wedge s_2 \in S_2\}$ and refer to this operation as the product of S_1 and S_2. We sometimes denote $S_1 \times ... \times S_n$ as $\bigtimes\{S_1, ..., S_n\}$.

Remark 2.9.12. Note that the neutral element with respect to the product operator is the set containing solely the empty set. In contrast, the product of the empty set and any other set of sets is the empty set itself, i.e.

For any set of sets S, $S \times \{\emptyset\} = S$ and $S \times \emptyset = \emptyset$.

2.10 Excerpts from graph theory

In this section, we briefly introduce all the notions of graph theory we are going to use later on in this report. Any graph G is assumed to be a directed graph unless stated otherwise.

For $G = (V, E)$, we denote $\{l \mid l \in V \wedge \nexists n \ s.t. \ (n \in V \wedge (l, n) \in E)\}$ as $leaves(G)$.

We sometimes refer to V as $nodes(G)$ and E as $edges(G)$.

$paths(G)$ stands for
$\{(n_1, ..., n_m) \mid \{n_1, ..., n_m\} \subseteq nodes(G) \wedge$
$\forall 1 \leq i \leq m - 1 \ (n_i, n_{i+1}) \in edges(G)\}$.

For nodes $\{n_1, n_2\} \in nodes(G)$, we define
$existsPath(n_1, n_2, G) \Leftrightarrow \exists p \in paths(G) \ s.t. \ (p = (n_1, ..., n_2))$.

We also define $depth(G) = \begin{cases} max\{ \mid p \mid \ \mid p \in paths(G)\} \text{ if } nodes(G) \geq 2 \\ 1 \text{ otherwise} \end{cases}$

Definition 2.10.1. A graph $G_S = (V_S, E_S)$ is an induced subgraph of $G = (V, E)$ iff

- $V_S \subseteq V$,
- $V_S \neq \emptyset$ and
- $E_S = \{(n_1, n_2) \mid \{n_1, n_2\} \subseteq V_S \wedge (n_1, n_2) \in E\}$

Definition 2.10.2. A graph $CC = (V_{CC}, E_{CC})$ is a connected component of $G = (V, E)$ iff

- CC is an induced subgraph of G and
- for any $v_i \in V$, if there exists $v_j \in V$ s.t. $v_j \in V_{CC}$ and either of the following three conditions hold:
  - $v_i = v_j$
  - $existsPath(v_i, v_j, CC)$
  - $existsPath(v_j, v_i, CC)$,

  then $v_i \in V_{CC}$

We denote the set $\{CC \mid \text{CC is a connected component of } G\}$ as $connComp(G)$.

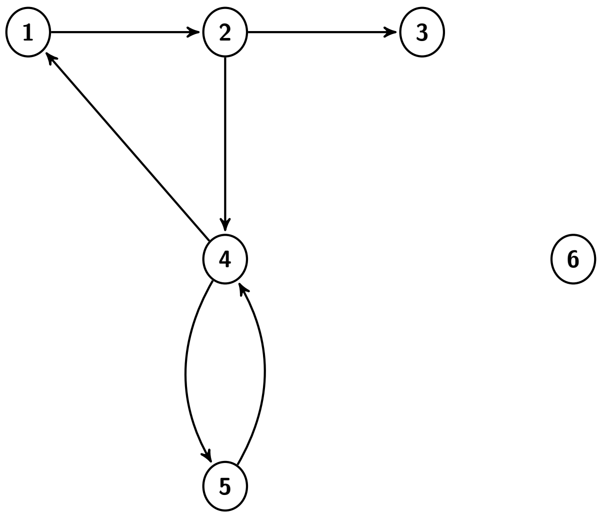
Definition 2.10.3. A graph $SCC = (V_{SCC}, E_{SCC})$ is a strongly connected component of $G = (V, E)$ iff

- SCC is an induced subgraph of G and
- for any $v_i \in V$, if there exists $v_j \in V$ s.t. $v_j \in V_{SCC}$ and either $v_i = v_j$ or both the following conditions hold:
  - $existsPath(v_i, v_j, CC)$
  - $existsPath(v_j, v_i, CC)$,

  then $v_i \in V_{SCC}$

Note that both connected and strongly connected components are maximal, i.e. no (strongly) connected component is contained within another (strongly) connected component.

Example 2.10.4. We wish to illustrate the difference between a connected and a strongly connected component by means of the following graph:



It has two connected components – the induced subgraph with nodes $1, 2, 3, 4, 5$ and the node 6. It has three strongly connected components – the induced subgraph with nodes $1, 2, 4, 5$ and the nodes 3 and 6.

3 Our approach to parallelizing ILP tasks

3.1 Theoretical Framework

For a given inductive learning task $< B, E, M >$ (as defined in section 2.9), we want to construct a dependency graph on B and M. Each of its edges represents a dependency between two predicates as defined below.

Definition 3.1.1. Let $< B, E, M >$ be an ILP task, S_M denote the skeleton rules corresponding to M, $typeBodyAtoms(r)$ – the set of body atoms of a skeleton rule $r \in S_M$ used to enforce the types and $nonTypeBodyAtoms(r) = body(r) \setminus typeBodyAtoms(r)$ (definition 2.9.5). A predicate p_1 is *dependent* on a predicate p_2 if either of the following conditions hold:

- $\exists r \in B \quad \text{s.t.} \quad p_1 = pred(head(r))$ and $p_2 \in pred(body(r))$
- $\exists r \in S_M \quad \text{s.t.} \quad p_1 = pred(head(r))$ and $p_2 \in pred(nonTypeBodyAtoms(r)) \vee p_2 \in pred(typeBodyAtoms(r)) \cap pred(M_h)$

Note that definition 3.1.1 captures only direct dependencies between two predicates. Transitive ones emerge as paths in the dependency graph. Furthermore, we are chiefly interested in the dependencies between head declaration predicates and thus leave out the non-head type predicates introduced by S_M.

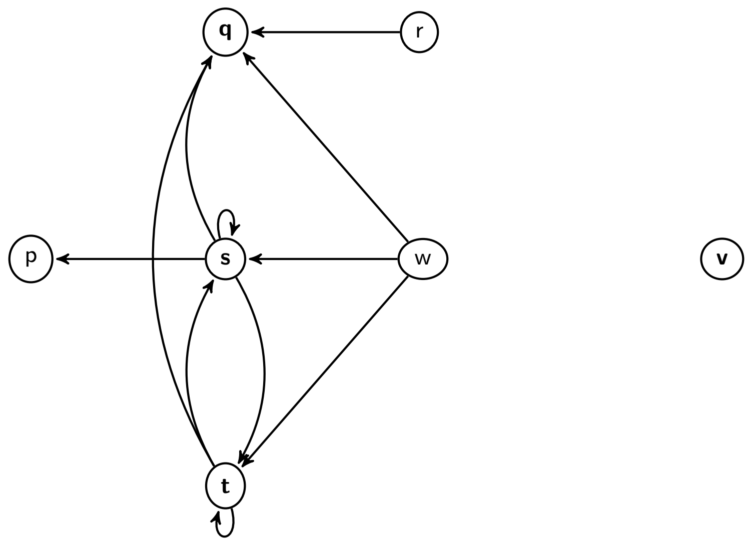
Definition 3.1.2. We call two sets of head declarations M_1 and M_2 disconnected iff $pred(M_1)$ and $pred(M_2)$ are disjoint and there is no predicate in any of $pred(M_1)$ and $pred(M_2)$ dependent on a predicate in the another.

We can now define the dependency graph and its reduction that we are going to use to split inductive learning tasks into smaller subtasks.

Definition 3.1.3. For an inductive learning task $< B, E, M >$, the dependency graph $depGr(B, M)$ is a directed graph (V, E), where:

- V is the set of head mode declaration predicates, body mode declaration predicates and background knowledge predicates
- $E = \{(p_2, p_1) \mid \{p_1, p_2\} \subseteq V$ and p_1 is dependent on p_2 according to definition 3.1.1 $\}$.

Example 3.1.4. The dependency graph of the inductive learning task
$< B, E, M >$ with
$B = \{p(X) \leftarrow s(X).\ t1(a).\ t1(e).\ t2(b).\ t2(d).\ t3(c).\ r(d, a).\ w(a).\}$,
$M_h = \{modeh(q(+t2)).\ modeh(s(+t1)).\ modeh(t(+t1)).modeh(v(+t3)).\}$,
$M_b = \{modeb(s(+t1)).\ modeb(t(+t1)).\ modeb(w(+t1)).\ modeb(r(+t2, -t1)).\}$,
$E^+ = \{p(a), q(d), t(a), v(c)\}$ and
$E^- = \{p(e), q(b)\}$
is the following:



Note that the predicates of nodes containing head declarations are printed in bold. The only edge representing a dependency through the background knowledge is (s, p). There are edges from any node in the set $S_1 = \{s, t, w\}$ to any node in the set $\{s, t\}$ because of the body declarations $B_1 = modeb(s(+t1))$. $modeb(t(+t1))$. $modeb(w(+t1))$. and the head declarations $modeh(s(+t1))$. $modeh(t(+t1))$., which are all specifying predicates with the same input variable types. Analogically, there is an edge from r to q because of the body declaration $modeb(r(+t2, -t1))$. and the head declaration $modeh(q(+t2))$., whose input variable types match. Because of the output variable type $-t1$ of predicate r and according to definition 2.9.5, any body atom of a skeleton rule with a head predicate q, which comes after an atom with predicate r, can be compatible with any body declaration with input type $+t1$. This applies to all of the body declarations B_1 and we therefore have an edge from each of the nodes in S_1 to q.

v is an isolated node because it does not appear in the background knowledge and there are no body declarations with input variable type $+t3$ to match its head declaration $modeh(v(+t3))$.

We now want to build a reduced dependency graph, each node of which corresponds to an ASPAL subtask. Predicates that form a cycle in the dependency graph must be learnt together, because there are circular dependencies between them. We therefore merge each strongly connected component of the dependency graph to a single node.

In order to enable non-observational predicate learning, we must ensure that all predicates, on which a predefined observed predicate depends, are learnt within the same ASPAL subtask. For that reason, the nodes containing those predicates are also merged together in the reduced dependency graph.

Each edge of the reduced dependency graph corresponds to a dependency between ASPAL subtasks. We remove all self-loops as trivial dependencies.

Definition 3.1.5. A graph $G_C = (V_C, E_C)$ is a non-observational component of $G = (V, E)$ with respect to an inductive learning task $< B, E, M >$ iff

- G_C is an induced subgraph of G
- $\forall n \in V_C\ \exists p \in V_C$ *s.t.* p is a predefined observed predicate in the sense of definition 2.9.4 and $n = p \vee (n, p) \in E_C$

Definition 3.1.6. The reduction $\mathit{redDepGr}(B, M)$ of $\mathit{depGr}(B, M)$ is the directed graph (V_r, E_r), where

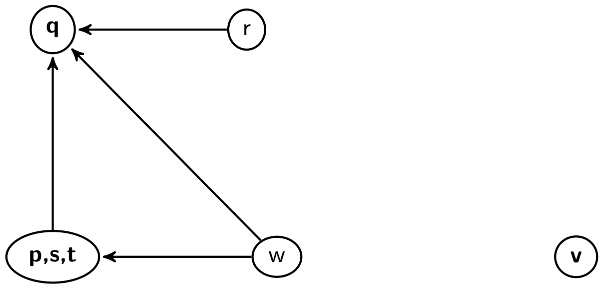
- V_r is the set of sets, each of which contains:
  - some nodes ns of $\mathit{depGr}(B, M)$ that form a strongly connected component in $\mathit{depGr}(B, M)$
  - the nodes that are part of a non-observational component in $\mathit{depGr}(B, M)$ containing any node in ns
- $E_r = \{(s_1, s_2) \mid s_1 \neq s_2 \wedge \{s_1, s_2\} \subseteq V_r \wedge \exists n_1 \in s_1\ \exists n_2 \in s_2\ s.t.\ (n_1, n_2) \in E\}$

Note that the reduced dependency graph is acyclic. If there were a cycle in the graph, we could have built a bigger strongly connected component of the dependency graph containing all the nodes of the strongly connected components in the cycle, which contradicts the maximality of a strongly connected component.

Also note that the connected components of the dependency graph and its reduction coincide, so that we can later use the former to construct the subtasks for the split in definition 3.2.3. We cannot, however, use a connected component of the dependency graph for the split in definition 3.2.4, so that we stick to the reduced dependency graph throughout.

An algorithm for the construction of the reduced dependency graph of a given inductive learning task is presented in the following section.

Example 3.1.7. Below, we present the reduction of the dependency graph from example 3.1.4:



Note that in the dependency graph from example 3.1.4, s and t belong to the same strongly connected component, whereas p is a predefined observed predicate forming a non-observational component together with s. They are therefore merged into a single node in the reduced dependency graph. All self-loops are removed and any remaining edges from or to any of the nodes in p, s, t in the dependency graph are replaced by new ones from or to the node p, s, t in the reduced dependency graph.

3.2 Method

Next, we want to formulate a split of an inductive learning task $< B, E, M >$ according to the reduced dependency graph from definition 3.1.6. In order to do so, we introduce some more notation.

Definition 3.2.1. For an inductive learning task $< B, E, M >$, denote the restriction of E to the examples of a set of predicates s as E_s. For an induced subgraph SG of $redDepGr(B, M)$, denote $\bigcup\{E_n \mid n \in nodes(SG)\}$ as E_{SG}.

Definition 3.2.2. For an inductive learning task $< B, E, M >$ and a head declaration predicate p, $mode(p) \subseteq M_h$ stands for the set of mode declarations with predicate p.

For a set of head declaration predicates s, define $mode(s)$ as the union of the mode declaration sets of all predicates in s, i.e. $mode(s) = \bigcup\{mode(p) \mid p \in s\}$.

For an induced subgraph SG of $redDepGr(B, M)$, define M_{SG} as the union of the mode declaration sets of all nodes in SG, i.e. $M_{SG} = \bigcup\{mode(n) \mid n \in nodes(SG)\}$.

Definition 3.2.3. We split the inductive learning task $< B, E, M >$ into the subtasks $\{< B, E_{CC}, < M_{CC}, M_b >) \mid CC \in connComp(redDepGr(B, M))\}$, the product of whose optimal solutions gives us $ASPAL^*(< B, E, M >)$.

The split from this definition is implemented in algorithm 3. A proof of its correctness is to be found in corollary 5.2.2.

Once we have split the given inductive learning task into subtasks corresponding to each connected component of the reduced dependency graph, we want to split further within each of the connected components. We therefore introduce an inductive definition for the rules of a connected component.

In the following, for a directed graph $G = (V, E)$ and a set of nodes $ns \subseteq V$, we write $remNs(G, ns)$ for $G' = (V', E')$ such that G' is an induced subgraph of G with $V' = V \setminus ns$.

Definition 3.2.4. For an inductive learning task $< B, E, M >$ and $CC \in connComp(redDepGr(B, M))$, define $CCRules(CC)$ as follows:

Base case: CC has a single node.
$CCRules(CC) = ASPAL^*(< B, E_{CC}, < M_{CC}, M_b >>)$

Inductive clause:
$CCRules(CC) = \{defComb \cup leafComb \mid$
$defComb \in \bigtimes\{CCRules(CSUBC) \mid$
$CSUBC \in connComp(remNs(CC, leaves(CC)))\} \wedge$
$leafComb \in \bigtimes\{ASPAL^*(< B \cup defComb, E_l,$
$< mode(l), M_b >>) \mid l \in leaves(CC)\}\}$

The variable $defComb$ stands for *defining rules combination*, i.e. a combination of the rules for each of the connected subcomponents $CSUBC$ once the leaves of the initial connected component has been removed, while $leafComb$ stands for *leaf rules combination*, i.e. a combination of rules for each of the leaves of the connected component CC.

Remark 3.2.5. Note that the definition of $CCRules(CC)$ in the base case coincides with its inductive clause given that CC has a single node, i.e.

$nodes(CC) = leaves(CC) \Rightarrow connComp(remNs(CC, leaves(CC))) = \emptyset \Rightarrow$

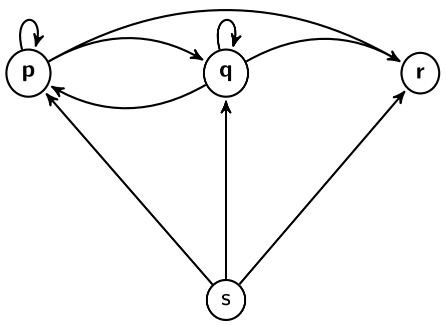
$defComb = \emptyset \Rightarrow CCRules(CC) = \bigtimes\{ASPAL^*(< B, E_l, < mode(l), M_b >>) \mid l \in nodes(CC)\} = ASPAL^*(< B, E_{CC}, < M_{CC}, M_b >>)$.

While the solution computation procedure within a connected component of the reduced dependency graph based on definition 3.2.4 is correct, it is not complete in respect to either general or optimal ASPAL solutions as shown in 5.2.5. Although rare in practise, it is even possible that $CCRules(CC)$ is empty even though $ASPAL(< B, E_{CC}, < M_{CC}, M_b >>)$ is not as in the example below.

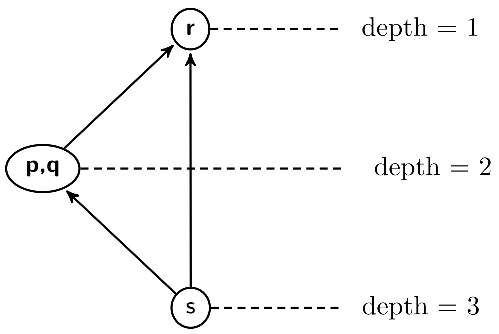
Example 3.2.6. Consider the following ASPAL program:

```
modeh(r(+t)).
modeh(p(+t)).
modeh(q(+t)).

modeb(p(+t)).
modeb(q(+t)).
modeb(s(+t)).

t(a).
t(b).
s(a).

r(X) :- p(X),q(X).

example(r(b),-1).
example(r(a),1).
example(p(a),1).
example(q(a),1).
```

Its dependency graph is the following:



Its reduced dependency graph has a single connected component CC as shown below.



At $depth = 1$, the only leaf contains the predicate r and there is one connected subcomponent with nodes $\{p, q\}$ and $\{s\}$. At $depth = 2$, this connected subcomponent has itself a single leaf with predicates p, q and its only connected subcomponent is the node $\{s\}$. s is not a head declaration predicate, the solution set for its connected subcomponent contains solely the empty set and thus `defComb` at $depth = 2$ is empty. `leafComb` at the same depth, on the other hand, contains the only solution S in $ASPAL^*(< B, E_{\{p,q\}}, < mode(\{p, q\}), M_b >>)$, which is the following:

```
q(A):- t(A).
p(A):- t(A).
```

It is also the `defComb` at $depth = 1$. Since $ASPAL^*(< B \cup S, E_{\{r\}}, < mode(r), M_b >>)$ is empty, there does not exist any `leafComb` at $depth = 1$ and thus $CCRules(CC)$ is empty. However, $ASPAL(< B, E_{CC}, < M_{CC}, M_b >>)$ contains, among others, the solution

```
q(A):- t(A), s(A).
p(A):- t(A).
```

Since for each split within a connected component of the reduced dependency graph, there is at least one predicate p such that p is dependent on q and the head declarations of p and q land into different subtasks after the split, we believe that a solution computation procedure complete in respect to optimal ASPAL solutions is infeasible. Below, we present a definition similar to 3.2.4 and complete in respect to general ASPAL solutions.

Definition 3.2.7. For an inductive learning task $< B, E, M >$ and $CC \in connComp(redDepGr(B, M))$, define $GeneralCCRules(CC)$ as follows:

Base case: CC *has a single node.*
$GeneralCCRules(CC) = ASPAL(< B, E_{CC}, < M_{CC}, M_b >>)$

Inductive clause:
$GeneralCCRules(CC) = \{defComb \cup leafComb \mid$
$defComb \in \bigtimes\{GeneralCCRules(CSUBC) \mid$
$CSUBC \in connComp(remNs(CC, leaves(CC)))\} \wedge$
$leafComb \in \bigtimes\{ASPAL(< B \cup defComb, E_l,$
$< mode(l), M_b >>) \mid l \in leaves(CC)\}\}$

Even though the solution computation procedure based on definition 3.2.7 is complete as opposed to the one based on definition 3.2.4, it is often infeasible in practice due to the huge number of general ASPAL solutions even for simple ILP tasks. The algorithm presented in the following section, which computes the solution set for a given connected component of the reduced dependency graph, is therefore based on definition 3.2.4.

4 Algorithm and implementation

4.1 Algorithm

We first need to define two auxiliary procedures – one for the construction of the reduced dependency graph of a given inductive learning task and another for the computation of a hypothesis set for each of its connected components.

Below, we present the algorithm for the construction of a reduced dependency graph as per definition 3.1.6 given background knowledge B and mode declarations $M =< M_h, M_b >$. Its correctness is proven in theorem 5.1.1. Each time it is called, the function `FindNewMatchingBodyDeclaration` $(predicate\_types, M_b)$ returns a new body mode declaration from M_b, whose input variable types are a subset of *predicate_types*, if such one is present and a null pointer otherwise.

Algorithm 1 Reduced Dependency Graph Construction

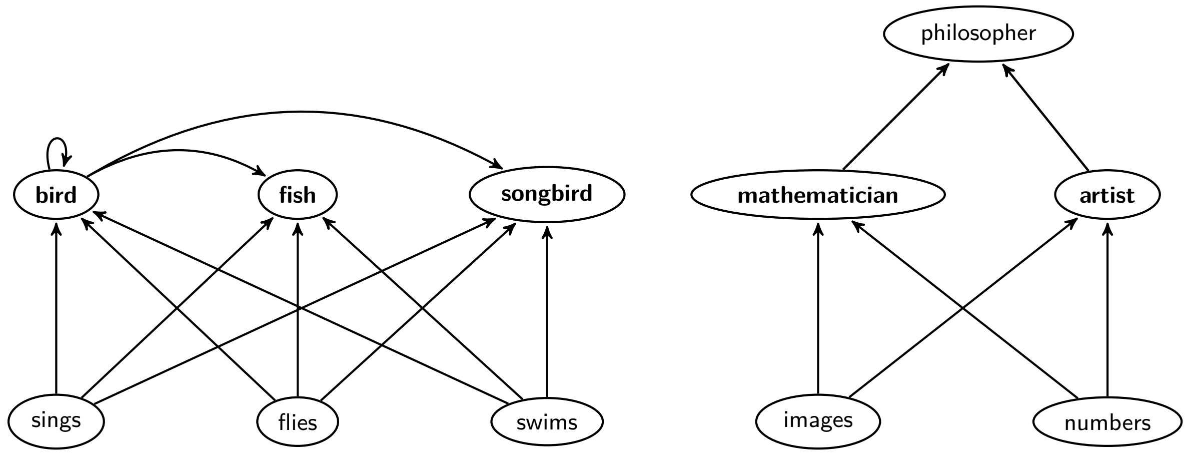
```
 1: procedure RedDepGr($B, M$)
 2:     $V \leftarrow pred(B) \cup pred(M)$
 3:     $E_S \leftarrow \emptyset$
 4:     for $modeh(h(+t_1, ..., +t_n)) \in M_h$ do
 5:         $predicate\_types \leftarrow \{t_1, ..., t_n\}$
 6:         for $t \in predicate_types$ do
 7:             if $t \in pred(M_h)$ then
 8:                 $E_S.add(t, h)$
 9:             end if
10:         end for
11:         $b \leftarrow$ findNewMatchingBodyDeclaration($predicate\_types, M_b$)
                //$b$ is of the form $modeb(b(+s_1, ..., +s_m, -s_{m+1}, ..., -s_l))$
12:         while $b \neq null$ do
13:             $E_S.add(b, h)$
14:             $predicate\_types.add([s_{m+1}, ..., s_l])$
15:             $b \leftarrow$ findNewMatchingBodyDeclaration($predicate\_types, M_b$)
16:         end while
17:     end for
18:     $E_B \leftarrow \emptyset$
19:     for $r \in B$ do
20:         for $b \in body(r)$ do
21:             $E_B.add(b, head(r))$
22:         end for
23:     end for
24:     $graph \leftarrow (V, E_S \cup E_B)$
25:     mergeNonObservationalComponents($graph$)
26:     mergeStronglyconnComp($graph$)
27:     removeSelfLoops($graph$)
28:     return $graph$
29: end procedure
```

Example 4.1.1. Let us consider the following ASPAL program:

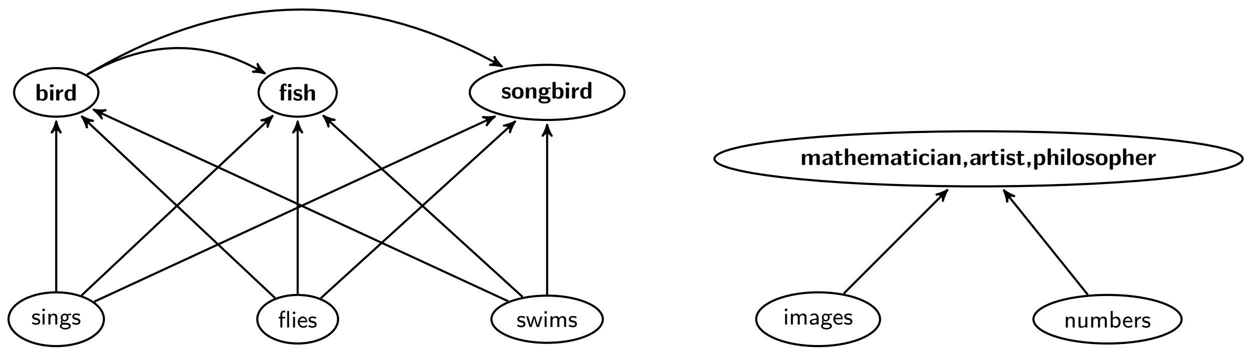
```
% mode declarations
```

```
modeh(artist(+human)).
modeh(mathematician(+human)).
modeh(bird(+animal)).
modeh(songbird(+animal)).
modeh(fish(+animal)).
modeb(images_affinity(+human)).
modeb(numbers_affinity(+human)).
modeb(bird(+animal)).
modeb(flies(+animal)).
modeb(swims(+animal)).
modeb(sings(+animal)).

% background knowledge
human(aa).
human(bb).
human(cc).
animal(a).
animal(b).
animal(c).
animal(d).
flies(a).
swims(b).
flies(c).
sings(c).
sings(d).
images_affinity(aa).
images_affinity(cc).
numbers_affinity(bb).
numbers_affinity(cc).
philosopher(X) :- artist(X), mathematician(X).

% examples
example(philosopher(aa),-1).
example(philosopher(bb),-1).
example(philosopher(cc),1).
example(artist(aa),1).
example(artist(bb),-1).
example(artist(cc),1).
example(bird(a),1).
example(bird(b),-1).
example(bird(c),1).
example(fish(a),-1).
example(fish(b),1).
example(songbird(a),-1).
example(songbird(b),-1).
example(songbird(c),1).
example(songbird(d),-1).
```

Its dependency graph according to definition 3.1.3 is the following (note that we have shortened the predicates *images_affinity* and *numbers_affinity* to *images* and *numbers*, respectively):



There is one non-observational component, consisting of the nodes *mathematician*, *artist* and *philosopher*. Each strongly connected component consists of a single node. The reduced dependency graph according to definition 3.1.6 is shown below:

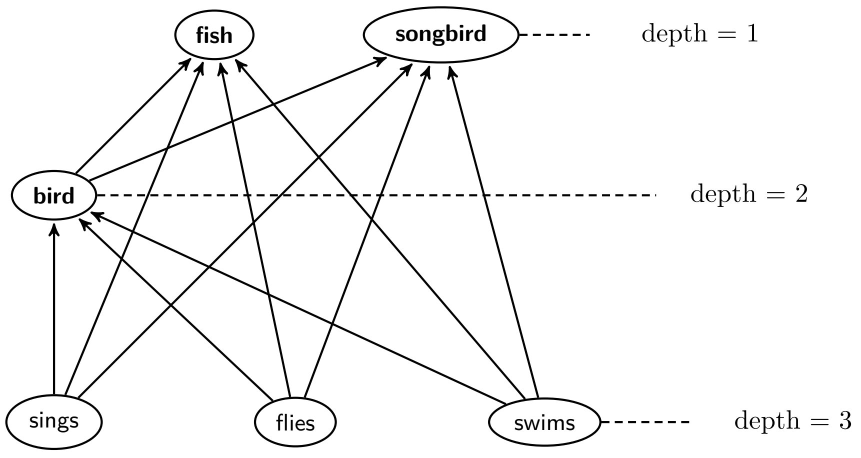


Next, we present the algorithm for the computation of ASPAL solutions for a given connected component *CC* of the reduced dependency graph, produced by algorithm 1. It complies with definition 3.2.4, while its correctness is proven in theorem 5.2.5.

Algorithm 2 Connected Component Rules

```
1: procedure CCRULES($CC$)
2:     if $\mid nodes(CC) \mid = 1$ then
3:         return $ASPAL^*(< B, E_{CC}, < M_{CC}, M_b >>)$
4:     else
5:         $component\_rules \leftarrow \emptyset$
6:         $subcomponent\_rules\_set \leftarrow \emptyset$
7:         for $CSUBC \in connComp(remNs(CC, leaves(CC)))$ do
8:             $subcomponent\_rules\_set.add(CCRules(CSUBC))$
9:         end for
10:        for $defining\_rules\_combination \in \bigtimes subcomponent\_rules\_set$ do
11:            $leave\_rules\_set \leftarrow \{defining\_rules\_combination\}$
12:            for $l \in leaves(CC)$ do
13:                $leave\_rules\_set.add(ASPAL^*$
                       $(< B \cup defining\_rules\_combination, E_l, < mode(l), M_b >>))$
14:            end for
15:            for $rule \in \bigtimes leave\_rules\_set$ do
16:                $component\_rules.add(rule)$
17:            end for
18:        end for
19:        return $component\_rules$
20:    end if
21: end procedure
```

Example 4.1.2. To illustrate the solution computing procedure within each of the connected components, consider the following connected component of the reduced dependency graph from example 4.1.1:



All of the nodes at $depth = 3$ do not contain any head declaration predicates and have the empty hypothesis as a solution, therefore the $defComb$ for $bird$ at $depth = 2$ is empty and has

```
bird(A):-animal(A), flies(A).
```

as a solution. This is also the $defComb$ for each of the nodes at $depth = 1$ with respective solutions

```
fish(A):-animal(A),swims(A).
```

for $fish$,

```
songbird(A):-animal(A), flies(A), sings(A).
```

and

```
songbird(A):-animal(A), bird(A), sings(A).
```

for *songbird*.

The set of solutions for the whole connected component is then the union of the product of the solution sets of $fish$ and *songbird* with their only *defining rules combination*, i.e.

```
fish(A):-animal(A),swims(A).
songbird(A):-animal(A), flies(A), sings(A).
bird(A):-animal(A), flies(A).
```

and

```
fish(A):-animal(A),swims(A).
songbird(A):-animal(A), bird(A), sings(A).
bird(A):-animal(A), flies(A).
```

The split of the original ASPAL task into ASPAL subtasks for each of the connected components of its reduced dependency graph is presented in the algorithm below, whose function calls `RedDepGr(`B, M`)` and `CCRules(`CC`)` refer to the procedures presented in algorithms 1 and 2. This split is based on definition 3.2.3. A proof of its correctness is to be found in corollary 5.2.2.

Algorithm 3 ASPAL Parallelizer

```
1: procedure PARALLELIZER(ASPAL input file)
2:     $< B, E, M > \leftarrow$ parseFile(ASPAL input file)
3:     $graph \leftarrow$ RedDepGr($B, M$)
4:     $component\_rules\_set \leftarrow \emptyset$
5:     for $CC \in connComp(graph)$ do
6:         $component\_rules\_set.add($CCRules(CC)$)$
7:     end for
8:     return $\bigtimes component\_rules\_set$
9: end procedure
```

Example 4.1.3. The reduced dependency graph from example 4.1.1 has two connected components, whose corresponding ASPAL subtasks have the following solutions:

Connected component with nodes {mathematician,artist,philosopher}, {images_affinity}, {numbers_affinity}:

```
mathematician(A):-human(A), numbers_affinity(A).
artist(A):-human(A), images_affinity(A).
```

Connected component with nodes {bird}, {fish}, {songbird}, {sings}, {flies}, {swims}: see example 4.1.2

The set of solutions of the original task is then exactly the product of the solution sets from above, i.e.

```
mathematician(A):-human(A), numbers_affinity(A).
artist(A):-human(A), images_affinity(A).
fish(A):-animal(A),swims(A).
songbird(A):-animal(A), bird(A), sings(A).
bird(A):-animal(A), flies(A).
```

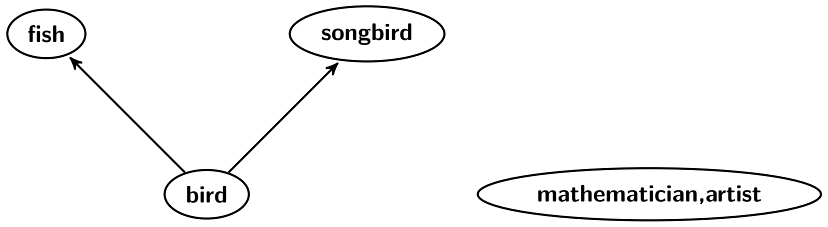
and

```
mathematician(A):-human(A), numbers_affinity(A).
artist(A):-human(A), images_affinity(A).
fish(A):-animal(A),swims(A).
songbird(A):-animal(A), flies(A), sings(A).
bird(A):-animal(A), flies(A).
```

4.2 Notes on the implementation

Our implementation is to be found in the file `parallelizer.py` and is supposed to be run in the same folder as `aspal.py`. A flag `--optimal` can be set so as to consider only the optimal solutions of each of the ASPAL subtasks in the construction of the final solution. Our code primarily follows algorithm 3.

Remark 4.2.1. We have noticed that the reduced dependency graph has many nodes not containing any head declaration predicates. Their corresponding ASPAL subtasks do not consequently include any head mode declarations, which means that the empty hypothesis is their only possible solution. Thus, they do not contribute to the final solution, but rather serve as a consistency check. As a final step in the reduced dependency graph construction in our implementation, we remove all nodes not containing at least one predicate in $pred(M_h)$ and add an edge from each of its immediate predecessors to each of its immediate successors. We then remove all predicates not in $pred(M_h)$ from the remaining nodes. In case the original learning task has examples of predefined observed predicates from any of the removed nodes, we create a new connected component, whose corresponding ASPAL subtask consists of the background knowledge and all these examples. The resulting graph of the ASPAL program from example 4.1.1 is for instance the following:



It can be proven that this adjustment of algorithm 3 does not affect the final solutions it produces, while improving its efficiency by substantially reducing the number of ASPAL subtasks.

Remark 4.2.2. Again from efficiency reasons, we have abstained from computing the strongly connected components of the dependency graph. Instead, we have iteratively merged nodes forming a simple cycle together. As a result, each node of the thus obtained graph corresponds to a strongly connected component of the original one.

We have also amended the implementation of ASPAL to tackle the following problems:

I. potentially wrong optimal solutions output in no-iteration mode

   We always run ASPAL in iteration mode when computing optimal solutions.

II. no difference in the output when no solutions exist as opposed to the case when the empty hypothesis is the only solution

   As we split the task into multiple subtasks, any of which might have an empty solution even if the original task has a non-empty one, the argument of the previous necessity of an ASPAL computation is rendered invalid. We therefore output the empty hypothesis as a member of the solution set whenever an empty answer set of the meta task is present and update the ASPAL statistics accordingly.

III. interference of multiple instances of ASPAL, running in parallel

We give a unique filename to each of the subtasks of the original task and then create a separate `workfile` for each instance of ASPAL, based on the input filename.

5 Proof of correctness

5.1 Theoretical framework

We first want to prove that the reduced dependency graph from definition 3.1.6 correctly represents the dependencies between the predicates of a given inductive learning task as specified in definition 3.1.1.

Lemma 5.1.1. Let $< B, E, M >$ be an ILP task and S_M denote the skeleton rules according to M, constructed with infinite number restrictions (definition 2.9.5). For each rule $r \in B \cup S_M$ with $dependent = pred(head(r))$ and $defining \in pred(nonTypeBodyAtoms(r)) \cup (pred(typeBodyAtoms(r)) \cap pred(M_h))$, there exists an edge $(s_1, s_2) \in edges(redDepGr(B, M))$ s.t. $defining \in s_1$ and $dependent \in s_2$. Moreover, there exist no further edges in $redDepGr(B, M)$.

Proof. We are going to base the proof on algorithm 1 and structure it as follows:

I. for the outer loop in line 4, which is responsible for the creation of edges representing dependencies according to the mode declarations, prove by induction an invariant stating that the set of edges at any time is a subset of the edges EDG postulated by the lemma and describing a dependency through the skeleton rules

   - include a condition for the current set of input variables into the invariant
   - prove by induction that the invariant holds for the inner loop in line 12
   - use the condition for the current set of input variables from the invariant in conjunction with the negated inner loop condition in order to prove the maximality of the set of edges to the head predicate last iterated upon in the inner loop
   - from this maximality and the invariant, conclude that all the edges in EDG to the head predicate last iterated upon in the inner loop coincide with the current set of edges to the same predicate

II. upon termination of the outer loop, conclude that the current set of edges coincides with EDG

III. prove that upon termination of the loop in line 19, which is responsible for the creation of edges representing dependencies according to the background knowledge, we have obtained the set of edges postulated by the lemma and describing a dependency through the background knowledge

IV. from points II. and III. and after execution of lines 24 – 27, conclude the correctness of the lemma

Note that for $r \in B$, $typeBodyAtoms(r) = \emptyset$ and $nonTypeBodyAtoms(r) = body(r)$. Since we have abstracted away the number restrictions on S_M, we only point to the first three bullet points of definition 2.9.5 in the proof of the lemma.

In the following, `V`, E_S, E_B, `h`, `t`, `b` and `predicate_types` refer to variables from the algorithm. For $h = modeh(p(+t_1, ..., +t_n)) \in M_h$, let $types(h) =$

$\{t_1, ..., t_n\}$. For $b = \mathit{modeb}(p(+s_1, ..., +s_m, -s_{m+1}, ..., -s_l))$, let $\mathit{types}(b) = \{s_1, ..., s_m\}$. Also let $|M_h| = n$.

I. We first prove an invariant for the loop in line 4:

(1) $\forall h \in \mathit{pred}(M_h)\ \forall s_1 \in \mathtt{V}\ (s_1, h) \in E_S \Rightarrow \exists r \in S_M\ \mathit{s.t.}\ h = \mathit{pred}(\mathit{head}(r)) \wedge s_1 \cap (\mathit{pred}(\mathit{nonTypeBodyAtoms}(r)) \cup (\mathit{pred}(\mathit{typeBodyAtoms}(r))) \cap \mathit{pred}(M_h))) \neq \emptyset$ and $\mathtt{predicate\_types} \subseteq \mathit{pred}(\mathit{types}(\mathtt{h})) \cup \{\mathit{pred}(\mathit{types}(b)) \mid b \in \mathit{pred}(M_b) \wedge \exists s \in \mathtt{V}\ \mathit{s.t.}\ b \in s \wedge (s, \mathtt{h}) \in E_S\}$.

We prove it by induction.

Base case: Trivially, it holds before we enter the loop as we have initialized E_S as the empty set in line 3.

Inductive assumption (IA_1): The invariant is true at the beginning of an arbitrary iteration of the loop.

Inductive step: Given (IA_1), $E_S \overset{\mathit{line}\ 8}{=} E_S \cup \{(\mathtt{t}, \mathtt{h})\}$ such that $\mathtt{h} \in \mathit{pred}(M_h)$, i.e. $\mathtt{h}$ respects the first bullet point of definition 2.9.5, and $\mathtt{t} \in \mathit{pred}(\mathit{types}(\mathtt{h})) \cap \mathit{pred}(M_h))$, i.e. $\mathtt{t}$ respects the third bullet point of definition 2.9.5, we obtain $\forall s_1 \in \mathtt{V}\ (s_1, \mathtt{h}) \in E_S \Rightarrow \exists r \in S_M\ \mathit{s.t.}\ h = \mathit{pred}(\mathit{head}(r)) \wedge s_1 \cap (\mathit{pred}(\mathit{nonTypeBodyAtoms}(r)) \cup (\mathit{pred}(\mathit{typeBodyAtoms}(r))) \cap \mathit{pred}(M_h))) \neq \emptyset$. We also have $\mathtt{predicate\_types} \overset{\mathit{line}\ 5}{=} \mathit{pred}(\mathit{types}(\mathtt{h})) \subseteq \mathit{pred}(\mathit{types}(\mathtt{h})) \cup \{\mathit{pred}(\mathit{types}(b)) \mid b \in \mathit{pred}(M_b) \wedge \exists s \in \mathtt{V}\ \mathit{s.t.}\ b \in s \wedge (s, \mathtt{h}) \in E_S\}$ and thus invariant is still true at the beginning of the inner loop in line 12. We then need to make another induction for the inner loop.

> *Base case*: See above.
>
> *Inductive assumption (IA_2)*: The invariant is true at the beginning of an arbitrary iteration of the loop.
>
> *Inductive step*: Given (IA_2) and $\mathtt{predicate\_types} \overset{\mathit{line}\ 14}{=} \mathtt{predicate\_types} \cup \mathit{pred}(\mathit{types}(\mathtt{b})) \overset{(*)}{\subseteq} \mathit{pred}(\mathit{types}(\mathtt{h})) \cup \{\mathit{pred}(\mathit{types}(b)) \mid b \in \mathit{pred}(M_b) \wedge \exists s \in \mathtt{V}\ \mathit{s.t.}\ b \in s \wedge (s, h) \in E_S\}$, as well as $E_S \overset{\mathit{line}\ 13}{=} E_S \cup \{(\mathtt{b}, \mathtt{h})\} \overset{(*)}{\Rightarrow}$
>
> $\forall s_1 \in \mathtt{V}\ (s_1, \mathtt{h}) \in E_S \Rightarrow \exists r \in S_M\ \mathit{s.t.}\ h = \mathit{pred}(\mathit{head}(r)) \wedge s_1 \cap (\mathit{pred}(\mathit{nonTypeBodyAtoms}(r)) \cup (\mathit{pred}(\mathit{typeBodyAtoms}(r))) \cap \mathit{pred}(M_h))) \neq \emptyset$
>
> (*) From the definition of `findNewMatchingBodyDeclaration` ($\mathit{predicate\_types}, M_b$) follows that the body predicate `b` it returns respects the second bullet point of definition 2.9.5. Since $\mathtt{h} \in \mathit{pred}(M_h)$, `h` respects the first bullet point of the same definition.

After the loop in line 12 has terminated, we use the invariant in conjunction with the fact that the termination of the loop is caused by `findNewMatchingBodyDeclaration(`$\mathit{predicate\_types}, M_b$`)` returning a null pointer. This means that we have a maximal set of edges pointing to h in the dependency graph. i.e.

(2) $\forall s_1 \in \mathtt{V}\ (s_1, \mathtt{h}) \in E_S \Leftarrow \exists r \in S_M\ \mathit{s.t.}\ h = \mathit{pred}(\mathit{head}(r)) \wedge s_1 \cap (\mathit{pred}(\mathit{nonTypeBodyAtoms}(r)) \cup (\mathit{pred}(\mathit{typeBodyAtoms}(r))) \cap \mathit{pred}(M_h))) \neq \emptyset$.

II. After the loop in line 4 has terminated, we can conclude that the $\textcircled{2}$ holds for each head mode declaration, which in conjunction with the invariant in $\textcircled{1}$ yields:

(3) $\forall h \in pred(M_h)\ \forall s_1 \in \mathtt{V}\ (s_1, h) \in E_S \Leftrightarrow \exists r \in S_M\ s.t.\ h = pred(head(r)) \wedge s_1 \cap (pred(nonTypeBodyAtoms(r)) \cup (pred(typeBodyAtoms(r))) \cap pred(M_h))) \neq \emptyset$.

III. Now that we have proven the correctness and completeness of the set of edges postulated by the lemma and describing a dependency through the skeleton rules, we continue our proof from line 19 onwards. It can be proven similarly to above that upon termination of the loop in line 19, we have

(4) $\quad (b, h) \in E_B \Leftrightarrow \exists r \in B\ s.t.\ b \in pred(body(r)) \wedge h = pred(head(r))$.

IV. From $\textcircled{3}$ and $\textcircled{4}$, we can conclude that

$\exists r \in B \cup S_M\ s.t.\ dependent = pred(head(r)) \wedge defining \in$
$pred(nonTypeBodyAtoms(r)) \cup (pred(typeBodyAtoms(r))) \cap pred(M_h))$
$\Leftrightarrow (defining, dependent) \in E_S \cup E_B$.

Finally, after the execution of lines 24 – 27, we obtain the property of the lemma. $\square$

5.2 Method

We want to prove the correctness of the split of an inductive learning task, introduced in definition 3.2.3. In order to do so, we prove a more general theorem, a corollary of which states the correctness of the split.

Theorem 5.2.1. For an inductive learning task $< B, E, M >$ and head declaration sets $\{h_1, ..., h_n\}$ to be split along, which are pairwise disconnected as per definition 3.1.2 and satisfy $\bigcup\{mode(h) \mid h \in h_1 \cup ... \cup h_n\} = M_h$,

$ASPAL^*(< B, E, M >) =$
$\bigtimes\{ASPAL^*(< B, E_h, < mode(h), M_b >>) \mid h \in \{h_1, ..., h_n\}\}$

The following proof follows the general outline of the proof of theorem 6.1.2, which is to be found in [9, chapter 3].

Proof. We prove the theorem by induction on the number of pairwise disconnected head declaration sets n.

Base case: n=0.
Trivial, because both sides of the equation are equal.

Inductive assumption (IA): $n = k \in \mathbb{N}_0$.
The equation above applies for $\{h_1, ..., h_k\}$.

Inductive step: $n = k + 1$, therefore we have to split along $\{h_1, ..., h_{k+1}\}$.
First, we make use of the inductive assumption to transform the right-hand side of the equation into:
$\bigtimes\{ASPAL^*(< B, E_h, < mode(h), M_b >>) \mid h \in \{h_1, ..., h_{k+1}\}\}$
$= ASPAL^*(< B, E_{h_{k+1}}, < mode(h_{k+1}), M_b >>) \times$
$\bigtimes\{ASPAL^*(< B, E_h, < mode(h), M_b >>) \mid h \in \{h_1, ..., h_k\}\}$
$\overset{(IA)}{=} ASPAL^*(< B, E_{h_{k+1}}, < mode(h_{k+1}), M_b >>) \times$
$ASPAL^*(< B, E_{h_1 \cup ... \cup h_k}, < mode(h_1) \cup ... \cup mode(h_k), M_b >>)$

$ASPAL^*(< B, E, M >)$ is by definition the set of optimal solutions of the abductive meta task corresponding to the inductive learning task $< B, E, M >$. Without loss of generality, we assume the abductive meta task is grounded. Each of its solutions can be seen as the union of hypotheses corresponding to groups of head declarations of the inductive task.

We start from an element of the left-hand side of the equation and apply a succession of equivalent transformations to obtain an element of its transformed right-hand side. Let $H^* \in ASPAL^*(< B, E, M >)$, $h \subseteq H^*$ denote the subset of the hypothesis H^* compatible with the set of head mode declarations $mode(h_{k+1})$ and $H \subseteq H^*$ – the subset of the hypothesis H^* compatible with the set of head mode declarations $mode(h_1) \cup ... \cup mode(h_k)$, so that $h \cup H = H^*$. Since h_{k+1} is disconnected from $h_1 \cup ... \cup h_k$, with h we can only prove the examples in $E_{h_{k+1}}$, whereas the ones in $E_{h_1 \cup ... \cup h_k}$ are to be proven by H.

Let $A^* \in AS(B \cup H^*)$ and $U_1 = HB(B \cup H)$. Since all head declarations in $\{h_1, ..., h_{k+1}\}$ are pairwise disconnected and $HB(B \cup H^*) = HB(B \cup H) \cup HB(h)$, $\forall r \in B \cup H^*$ $head(r) \in U_1 \Rightarrow atoms(r) \not\subset HB(h) \Rightarrow atoms(r) \subset U_1$ and therefore U_1 is a splitting set of $B \cup H^*$. According to the splitting set theorem, there are X and Y so that $A^* = X \cup Y$, $X \cap Y = \emptyset$, X is an answer set of $bot_{U_1}(B \cup H^*) = B \cup H$ and Y – an answer set of $e_{U_1}(top_{U_1}(B \cup H^*), X) = e_{U_1}(h, X)$.

Let $U_2 = HB(B)$. Since none of the head declaration predicates is defining for any of the background knowledge predicates and $HB(B \cup H) = HB(B) \cup$

$HB(H)$, $\forall r \in B \cup H$ $head(r) \in U_2 \Rightarrow atoms(r) \not\subset HB(H) \Rightarrow atoms(r) \subset U_2)$ and therefore U_2 is a splitting set of $B \cup H$. According to the splitting set theorem, there are X_1 and X_2 so that $X = X_1 \cup X_2$, $X_1 \cap X_2 = \emptyset$, X_1 is an answer set of $bot_{U_2}(B \cup H) = B$ and X_2 – an answer set of $e_{U_2}(top_{U_2}(B \cup H), X_1) = e_{U_2}(H, X_1)$.

Considering that $\{h_1, ..., h_{k+1}\}$ are pairwise disconnected, $HB(h) \cap HB(H) = \emptyset$ and $Y \in AS(e_{U_1}(h, X)) \Leftrightarrow Y \in AS(e_{U_1}(h, X_1 \cup X_2)) \Leftrightarrow Y \in AS(e_{U_1}(h, X_1))$. Since $Y \cap X_1 = \emptyset, Y \in AS(e_{U_1}(h, X_1)), X_1 \in AS(B) = AS(bot_{U_1}(B \cup h))$ and U_1 is a splitting set for $B \cup h$, we can apply the splitting set theorem in the reverse direction to obtain an answer set $S = X_1 \cup Y$ of $B \cup h$. Thus $h \in ASPAL^*(< B, E_{h_{k+1}}, < mode(h_{k+1}), M_b >>)$. We already know that X is an answer set of $B \cup H$, which means that $H \in ASPAL^*(< B, E_{h_1 \cup ... \cup h_k},$
$< mode(h_1) \cup ... \cup mode(h_k), M_b >>)$, and thus obtain the equivalence
$h \in ASPAL^*(< B, E_{h_{k+1}}, < mode(h_{k+1}), M_b >>) \wedge$
$H \in ASPAL^*(< B, E_{h_1 \cup ... \cup h_k}, < mode(h_1) \cup ... \cup mode(h_k), M_b >>) \Leftrightarrow$
$h \cup H \in ASPAL^*(< B, E_{h_1 \cup ... \cup h_{k+1}}, < mode(h_1) \cup ... \cup mode(h_{k+1}), M_b >>)$ □

The following corollary states the correctness of definition 3.2.3.

Corollary 5.2.2. For a inductive learning task $< B, E, M >$,

$ASPAL^*(< B, E, M >) =$
$\bigtimes\{ASPAL^*(< B, E_{CC}, < M_{CC}, M_b >) \mid CC \in connComp$
$(redDepGr(B, M))\}$

Proof. Any two head declaration sets M_1, M_2 from
$\{M_{CC} \mid CC \in connComp(redDepGr(B, M))\}$ contain nodes from different connected components of the reduced dependency graph, therefore there is no edge between any pair p_1, p_2 such that $p_1 \in pred(M_1)$ and $p_2 \in pred(M_2)$. According to lemma 5.1.1, this is equivalent to there being no dependency between p_1 and p_2. Then by definition 3.1.2, the head declaration sets $\{M_{CC} \mid CC \in connComp(redDepGr(B, M))\}$ are pairwise disconnected.

The corollary is therefore a special case of theorem 5.2.1 with inductive task $< B, E, M >$ and pairwise disconnected head declaration sets
$\{M_{CC} \mid CC \in connComp(redDepGr(B, M))\}$ to be split along. □

Remark 5.2.3. Note that in both theorem 5.2.1 and its corollary, as well as in their proof, the sets of optimal ASPAL solutions as specified in definition 2.9.9 can be consistently replaced by the set of general ASPAL solutions as per definition 2.9.8.

We now want to prove the correctness of the split within a given connected component of the reduced dependency graph, as specified in definition 3.2.4. We first formulate a lemma that we are going to use in the correctness proof of the split in a non-base case.

Lemma 5.2.4. For a connected component CC of $redDepGr(B, M)$ of an inductive learning task $< B, E, M >$,

$\{defComb \cup leafComb \mid$
$defComb \in \bigtimes\{ASPAL(< B, E_{CSUBC}, < M_{CSUBC}, M_b >>) \mid$
$CSUBC \in connComp(remNs(CC, leaves(CC)))\} \wedge$
$leafComb \in \bigtimes\{ASPAL^*(< B \cup defComb, E_l, < mode(l), M_b >>) \mid$
$l \in leaves(CC)\}\} \subseteq ASPAL(< B, E_{CC}, < M_{CC}, M_b >>)$.

Proof. We first show that $\bigtimes\{ASPAL^*(< B \cup defComb, E_l,$
$< mode(l), M_b >>) \mid l \in leaves(CC)\} =$
$ASPAL^*(< B \cup defComb, E_{CC}, < M_{leaves(cc)}, M_b >>)$.

This is a special case of theorem 5.2.1 with inductive task
$< B \cup defComb, E_{leaves(cc)}, < M_{leaves(cc)}, M_b >>$ and pairwise disconnected head declaration sets $\{mode(l) \mid l \in leaves(cc)\}$ to be split along.

Secondly, we show that $\bigtimes\{ASPAL(< B, E_{CSUBC}, < M_{CSUBC}, M_b >>) \mid$
$CSUBC \in connComp(remNs(CC, leaves(CC)))\} =$
$ASPAL(< B, E_{remNs(CC,leaves(CC))}, < M_{remNs(CC,leaves(CC))}, M_b >>)$.

This is a special case of theorem 5.2.1 with inductive task
$< B, E_{remNs(CC,leaves(CC))}, < M_{remNs(CC,leaves(CC))}, M_b >>$
and pairwise disconnected head declaration sets
$\{mode(n) \mid n \in remNs(CC, leaves(CC))\}$ to be split along.

Lastly, we show that $\{defComb \cup leafComb \mid defComb \in$
$ASPAL(< B, E_{remNs(CC,leaves(CC))}, < M_{remNs(CC,leaves(CC))}, M_b >>) \wedge$
$leafComb \in ASPAL^*(< B \cup defComb, E_{CC}, < M_{leaves(cc)}, M_b >>)\} \subseteq$
$ASPAL(< B, E_{CC}, < M_{CC}, M_b >>)$.

We prove that each element of the left-hand side is an element of the right-hand side. Since no node within the connected component is dependent on a leaf, with $leafComb$ we can only prove the examples in $E_{leaves(CC)}$, whereas the ones in $E_{remNs(CC,leaves(CC))}$ are to be proven by $defComb$. For $defComb \in$
$ASPAL(< B, E_{remNs(CC,leaves(CC))},$
$< M_{remNs(CC,leaves(CC))}, M_b >>)$ and $U = HB(B \cup defComb)$, let $leafComb \in$
$ASPAL^*(< B \cup defComb, E_{CC}, < M_{leaves(cc)}, M_b >>)$.
Also let $H^* = defComb \cup leafComb$.

Since no node within the connected component is dependent on a leaf, U is a splitting set for $B \cup H^*$ with $bot_U(B \cup H^*) = B \cup defComb$ and $top_U(B \cup H^*) = leafComb$. Let $defCombAS \in AS(B \cup defComb) = AS(bot_U(B \cup H^*))$ and $leafCombAS \in e_U(leafComb, defCombAS) = e_U(top_U(B \cup H^*), defCombAS)$. Since U is a splitting set, $defCombAS \cap leafCombAS = \emptyset$.

According to the splitting set theorem applied in the reverse direction, $A^* = defCombAS \cup leafCombAS \in AS(B \cup H^*)$, wherefore $H^* \in ASPAL(< B, E_{CC}, < M_{CC}, M_b >>)$.

We have thus obtained $defComb \in ASPAL(< B, E_{remNs(CC,leaves(CC))},$
$< M_{remNs(CC,leaves(CC))}, M_b >>) \wedge$
$leafComb \in ASPAL^*(< B \cup defComb, E_{CC}, < M_{leaves(cc)}, M_b >>) \Rightarrow$
$defComb \cup leafComb \in ASPAL(< B, E_{CC}, < M_{CC}, M_b >>)$. □

Theorem 5.2.5. For an ILP task $< B, E, M >$ and a connected component $CC \in connComp(redDepGr(B, M))$,

$CCRules(CC) \subseteq ASPAL(< B, E_{CC}, < M_{CC}, M_b >>)$.

Note that CC is a connected component of the reduced dependency graph $redDepGr(B, M)$ (definition 3.1.6) of the ILP task $< B, E, M >$, E_{CC} denotes the examples of predicates in CC as per definition 3.2.1, M_{CC} – the head declarations of predicates in CC as per definition 3.2.2 and $ASPAL(< B, E_{CC}, < M_{CC}, M_b >>)$ – the general solution set (definition 2.9.8) of the ASPAL task $< B, E_{CC}, < M_{CC}, M_b >>$.

Proof. Inductive proof of the theorem: Induction on the depth of the graph n = depth(CC).

Base case: $n = 1$.
Since the depth of the connected component equals one, it has a single node. The statement of the theorem thus coincides with the base case of definition 3.2.4.

Inductive assumption(IA): $1 \leq n \leq k, k \in \mathbb{N}$.
For a connected component CC with $1 \leq depth(CC) \leq k$, $CCRules(CC) \subseteq ASPAL(< B, E_{CC}, < M_{CC}, M_b >>)$.

Inductive step: $n = k + 1$.

$$\begin{aligned}
&CCRules(CC) = \{defComb \cup leafComb \mid \\
&defComb \in \bigtimes\{CCRules(CSUBC) \mid \\
&CSUBC \in connComp(remNs(CC, leaves(CC)))\} \wedge \\
&leafComb \in \bigtimes\{ASPAL^*(< B \cup defComb, E_l, \\
&< mode(l), M_b >>) \mid l \in leaves(CC)\}\} \\
&\overset{(IA)\wedge depth(CSUBC)\leq k}{\subseteq} \{defComb \cup leafComb \mid \\
&defComb \in \bigtimes\{ASPAL(< B, E_{CSUBC}, < M_{CSUBC}, M_b >>) \mid \\
&CSUBC \in connComp(remNs(CC, leaves(CC)))\} \wedge \\
&leafComb \in \bigtimes\{ASPAL^*(< B \cup defComb, E_l, \\
&< mode(l), M_b >>) \mid l \in leaves(CC)\}\} \\
&\overset{Lemma\ 5.2.4}{\subseteq} ASPAL(< B, E_{CC}, < M_{CC}, M_b >>).
\end{aligned}$$
□

Note that in both lemma 5.2.4 and theorem 5.2.5, we obtain a subset of the *general* ASPAL solutions for the original task, even though we compute only *optimal* ASPAL solutions for each of the subtasks. If we consider general ASPAL solutions throughout, the completeness of the solution computing procedure based on definition 3.2.7 can be shown.

Lemma 5.2.6. For a connected component CC of $redDepGr(B, M)$ of an inductive learning task $< B, E, M >$,

$$\begin{aligned}
&\{defComb \cup leafComb \mid \\
&defComb \in \bigtimes\{ASPAL(< B, E_{CSUBC}, < M_{CSUBC}, M_b >>) \mid \\
&CSUBC \in connComp(remNs(CC, leaves(CC)))\} \wedge \\
&leafComb \in \bigtimes\{ASPAL(< B \cup defComb, E_l, < mode(l), M_b >>) \mid \\
&l \in leaves(CC)\}\} = ASPAL(< B, E_{CC}, < M_{CC}, M_b >>).
\end{aligned}$$

Proof. We first show that $\bigtimes\{ASPAL(< B \cup defComb, E_l,$
$< mode(l), M_b >>) \mid l \in leaves(CC)\} =$
$ASPAL(< B \cup defComb, E_{CC}, < M_{leaves(cc)}, M_b >>)$.

This is a special case of theorem 5.2.1 with inductive task $< B \cup defComb, E_{leaves(cc)}, < M_{leaves(cc)}, M_b >>$ and pairwise disconnected head declaration sets $\{mode(l) \mid l \in leaves(cc)\}$ to be split along.

Secondly, we show that $\bigtimes\{ASPAL(< B, E_{CSUBC}, < M_{CSUBC}, M_b >>) \mid$ $CSUBC \in connComp(remNs(CC, leaves(CC)))\} =$ $ASPAL(< B, E_{remNs(CC,leaves(CC))}, < M_{remNs(CC,leaves(CC))}, M_b >>)$.

This is a special case of theorem 5.2.1 with inductive task $< B, E_{remNs(CC,leaves(CC))}, < M_{remNs(CC,leaves(CC))}, M_b >>$ and pairwise disconnected head declaration sets $\{mode(n) \mid n \in remNs(CC, leaves(CC))\}$ to be split along.

Lastly, we show that $\{defComb \cup leafComb \mid defComb \in$ $ASPAL(< B, E_{remNs(CC,leaves(CC))}, < M_{remNs(CC,leaves(CC))}, M_b >>) \wedge$ $leafComb \in ASPAL(< B \cup defComb, E_{CC}, < M_{leaves(cc)}, M_b >>)\} =$ $ASPAL(< B, E_{CC}, < M_{CC}, M_b >>)$.

We start from an element of the left-hand side of the equation and apply a succession of equivalent transformations to obtain an element of its right-hand side.

Let $H^* \in ASPAL(< B, E_{CC}, < M_{CC}, M_b >>)$, $leafComb \subseteq H^*$ denote the subset of the hypothesis H^* compatible with the set of head mode declarations $M_{leaves(cc)}$ and $defComb = H^* \setminus leafComb$. Since no node within the connected component is dependent on a leaf, with $leafComb$ we can only prove the examples in $E_{leaves(CC)}$, whereas the ones in $E_{remNs(CC,leaves(CC))}$ are to be proven by $defComb$.

Let $A^* \in AS(B \cup H^*)$ and $U = HB(B \cup defComb)$. Since no node within the connected component is dependent on a leaf and $HB(B \cup H^*) =$ $HB(B \cup defComb) \cup HB(leafComb)$, $\forall r \in B \cup H^*$ $head(r) \in U \Rightarrow atoms(r) \not\subset$ $HB(leafComb) \Rightarrow atoms(r) \subset U$ and therefore U is a splitting set of $B \cup H^*$. According to the splitting set theorem, there are X and Y so that $A^* = X \cup Y$, $X \cap Y = \emptyset$, X is an answer set of $bot_U(B \cup H^*) = B \cup defComb$ and Y – an answer set of $e_U(top_U(B \cup H^*), X) = e_U(leafComb, X)$. Therefore $defComb \in$ $ASPAL(< B, E_{remNs(CC,leaves(CC))}, < M_{remNs(CC,leaves(CC))}, M_b >>)$ and $leafComb \in ASPAL(< B \cup defComb, E_{CC}, < M_{leaves(cc)}, M_b >>)$

We have thus obtained the equivalence
$leafComb \in ASPAL(< B \cup defComb, E_{CC}, < M_{leaves(cc)}, M_b >>) \wedge$
$defComb \in ASPAL(< B, E_{remNs(CC,leaves(CC))},$
$< M_{remNs(CC,leaves(CC))}, M_b >>) \Leftrightarrow$
$leafComb \cup defComb \in ASPAL(< B, E_{CC}, < M_{CC}, M_b >>)$ $\square$

Theorem 5.2.7. For an ILP task $< B, E, M >$ and a connected component $CC \in connComp(redDepGr(B, M))$,

$GeneralCCRules(CC) = ASPAL(< B, E_{CC}, < M_{CC}, M_b >>)$

Proof. Inductive proof of the theorem: Induction on the depth of the graph n = depth(CC).

Base case: $n = 1$.
Since the depth of the connected component equals one, it has a single node. The statement of the theorem thus coincides with the base case of definition 3.2.7.

Inductive assumption(IA): $1 \leq n \leq k, k \in \mathbb{N}$.
For a connected component CC with $1 \leq depth(CC) \leq k$, $GeneralCCRules(CC) =$ $ASPAL(< B, E_{CC}, < M_{CC}, M_b >>)$.

Inductive step: $n = k + 1$.
$GeneralCCRules(CC) = \{defComb \cup leafComb \mid$
$defComb \in \bigtimes\{GeneralCCRules(CSUBC) \mid$
$CSUBC \in connComp(remNs(CC, leaves(CC)))\} \wedge$
$leafComb \in \bigtimes\{ASPAL(< B \cup defComb, E_l,$
$< mode(l), M_b >>) \mid l \in leaves(CC)\}\}$

$\overset{(IA) \wedge depth(CSUBC) \leq k}{=} \{defComb \cup leafComb \mid$
$defComb \in \bigtimes\{ASPAL(< B, E_{CSUBC}, < M_{CSUBC}, M_b >>) \mid$
$CSUBC \in connComp(remNs(CC, leaves(CC)))\} \wedge$
$leafComb \in \bigtimes\{ASPAL(< B \cup defComb, E_l,$
$< mode(l), M_b >>) \mid l \in leaves(CC)\}\}$
$\overset{Lemma\ 5.2.6}{=} ASPAL(< B, E_{CC}, < M_{CC}, M_b >>).$ $\square$

6 Related work

In this section, we first give a brief summary of previous efforts at overcoming the grounding issue of ASPAL and then compare them with our approach from section 3.

6.1 PASPAL

A parallel logic-based system called PASPAL was introduced in [9]. The idea is to split a given ASPAL task into multiple ones, which serve as an input for ASPAL and are then post-processed to obtain the solution of the original task. A parallelization algorithm for a subset of ASPAL tasks was formulated and its soundness proven. The condition that all ASPAL tasks in this subset have to fulfill is to only contain disconnected head declarations as defined below.

Definition 6.1.1. Let $< B, E, M >$ be an inductive learning task. A head declaration $h = modeh(p(g_1, ..., g_n)) \in M_h$ is connected iff $\exists b = modeb(q(h_1, ..., h_m)) \in M_b$ *s.t.* $p = q \vee p = \sim q$ or there exists a rule $r \in B$ such that $atoms(r)$ contains an atom compatible with h. It is disconnected otherwise.

Theorem 6.1.2. Let $< B, E, M >$ be an inductive learning task, where all head declarations $M_h = \{h_1, ..., h_n\}$ are disconnected. Then:

$$\begin{aligned}&ASPAL(< B, E, M >) = \\ &ASPAL(< B, < E^+_{h_1}, E^- >, < \{h_1\}, M_b >>) \times \\ &... \qquad\qquad\qquad\qquad\qquad\qquad \times \\ &ASPAL(< B, < E^+_{h_n}, E^- >, < \{h_n\}, M_b >>),\end{aligned}$$

where E^+_h stands for the restriction of the positive examples to those compatible with h.

The proof can be found in [9, chapter 3]. An algorithmic implementation based on the splitting from the theorem is also presented there.

Example 6.1.3. An inductive logic program P that can be solved by means of PASPAL is for instance the following (taken from [9, chapter 3]).

```
modeh(penguin(+bird)). % not in P_reptile
modeb(not can(+bird, #ability)).

modeh(reptile(+animal)). % not in P_penguin
modeb(can(+animal, #ability)).

ability(fly).
ability(swim).
bird(a).
bird(b).
can(a, fly).
can(b, swim).
animal(p).
animal(q).
can(p, fly).
can(q, swim).
```

```
example(penguin(a),-1).
example(penguin(b),1). % not in P_reptile

example(reptile(p),-1).
example(reptile(q),1). % not in P_penguin
```

After splitting P according to theorem 6.1.2, we obtain the programs $P_{penguin}$ and $P_{reptile}$, constructed by omitting the respective lines marked in the comments. ASPAL computes

```
penguin(X):- bird(X), not can(X,fly).
```

to be the sole solution of $P_{penguin}$ and

```
reptile(X):- animal(X), can(X,swim).
```

– the only solution of $P_{reptile}$. The solution of P is then the product of both, i.e. in this case their union.

Remark 6.1.4. In case at least one connected head declaration according to definition 6.1.1 is detected, PASPAL passes the given ILP task unchanged to ASPAL as stated in [9, chapter 4].

6.2 General case with connected predicates

This section introduces a general approach dealing with any given ILP task, especially one with connected head predicates. It is a summary of the key points from [9, chapter 7].

Definition 6.2.1. Let $M_c \subseteq M_h$ denote the subset of all connected head declarations of a given inductive learning task $< B, E, M >$. The task $< B, E, < M_c, M_b >>$ is called the bottom task with respect to the original ASPAL task.

Based on the notion of a bottom task, the following split was proposed for any inductive learning task.

Theorem 6.2.2. Let $< B, E, M >$ be an inductive learning task and H_{bottom} denote the set of ASPAL solutions of the associated bottom task. Then we have:

$$\begin{array}{ll} ASPAL(< B, E, M >) = & \\ \{h & \cup \\ ASPAL(< B \cup h, < E^+_{i_1}, E^- >, < \{i_1\}, M_b >>) & \times \\ ... & \times \\ ASPAL(< B \cup h, < E^+_{i_m}, E^- >, < \{i_m\}, M_b >>) & \\ \mid h \in H_{bottom}\}, & \end{array}$$

where E^+_h stands for the restriction of the positive examples to those compatible with h and $\{i_1, ..., i_m\} = M_h \setminus M_c$.

The most significant problem with the bottom task is that it contains the whole set of examples of the original ILP task. This means it does not have any solutions in case not all head declarations are connected and there is at least one example of a predicate from a disconnected head declaration not already covered by the background knowledge. It is therefore sensible to restrict the examples of the bottom task to the ones of predicates in M_c. It is clear that the split from theorem 6.1.2 is subsumed by the corrected split from theorem 6.2.2. Even with the latter, we would not be able to achieve the fine granularity of the joint split from the definitions 3.2.3 and 3.2.4, as expounded in section 6.4.

6.3 RASPAL

Another approach to improve the scalability of ASPAL, RASPAL, aims directly at solving the grounding issue of the abductive meta task (see [10]). It employs theory revision on partial hypotheses at each iterative refinement step until it has constructed a solution of the given ILP task.

Since no program implementing RASPAL is publicly available yet, we were not able to measure its runtime and set it against the runtime of our approach in section 7. Because its underlying principle is so different from ours, we did not think a comparison of both would be sensible. We have instead proposed a hybrid approach combining an initial split done by our Parallelizer with RASPAL instead of ASPAL for the solution of each of the resulting subtasks (section 8.3).

6.4 Comparison of previous milestones with our approach

One of the crucial advantages of our approach over the ones from theorems 6.1.2 and 6.2.2 is that it relies on the binary relation of dependency from definition 3.1.1 rather than on the unary relation of connectedness from definition 6.1.1. The dependency relation then forms the basis of the dependency graph (definition 3.1.3), which is a rigorous formalization of the dependencies between predicates of a given ILP task. In its reduction (definition 3.1.6), each of the nodes is a set of predicates, which belong together in the split to come due to circular dependencies or as part of a non-observational component (definition 3.1.5).

Based on the connected components of the reduced dependency graph, we were able to identify the disconnected sets of predicates of the ILP task and create a subtask for each of them following the split from definition 3.2.3. This is a natural continuation of the idea from theorem 6.1.2, in that it is generalized for disconnected sets of predicates instead of disconnected head declarations. It is thereby also complete with respect to all ILP tasks.

Another advantage of our approach is that the split from definition 3.2.3 is paired with another split within each of the connected components of the reduced dependency graph (definition 3.2.4). The underlying idea of this split is inspired by 6.2.2 and takes it further in applying the notion of bottom tasks instead of a single bottom task. The split is also inductive, which enables us to create a separate subtask for each node of the reduced dependency graph and run all subtasks at the same depth in parallel once we have computed the solutions of all subtasks at lower depths.

A further benefit of our approach is that by analyzing the connected components of the dependency graph, we account for the transitivity of the dependency relation. We can thus identify both direct and indirect dependencies between predicates and create several subtasks containing predicates of the bottom task (definition 6.2.1) in case there is not a direct or indirect dependency between each pair of them.

Example 6.4.1. The inductive learning task from example 4.1.1 is not splittable with theorem 6.1.2 since the connected head declarations for predicates *bird* , *mathematician* and *artist* are present. According to theorem 6.2.2, we would only be able to split the task into subtasks for the following sets of predicates: {bird,mathematician,artist}, {fish} and {songbird}.

Based on the reduced dependency graph, however, we were able to identify two disconnected sets of predicates, which form separate subtasks according to definition 3.2.3. Further splitting of each of these subtasks according to definition 3.2.4 provides us with a separate subtask for each of the nodes of the graph, as outlined in example 4.1.2. The simplified reduced dependency graph we use in the actual implementation as shown in remark 4.2.1 eventually determines that with our approach, we obtain subtasks for the following sets of predicates: {bird}, {fish}, {songbird} and {mathematician,artist}.

7 Evaluation

In order to show that our algorithm 3 subsumes PASPAL, whose underlying mechanism is described in section 6.1, we ran it over the mobile dataset analyzed in [9, chapter 5]. Below, we list the mode declarations we have used to define our ILP task.

```
% head declarations
modeh(accept(+date,+contact,+volume,+vibrator,
  +battery_level,+screen_brightness,+light_level,+battery_charging)).

modeh(accept(+date,#contact,+volume,+vibrator,
  +battery_level,+screen_brightness,+light_level,+battery_charging)).

modeh(busy(+date,+time,+volume,+vibrator,+battery_level,
  +screen_brightness,+light_level,+battery_charging)).

modeh(priority_contact(#contact)).

% body declarations
modeb(user_is_using_app(+date, +time, #app)).
modeb(not user_is_using_app(+date, +time, #app)).
modeb(high_volume(+volume)).
modeb(not high_volume(+volume)).
modeb(low_battery(+battery_level)).
modeb(not low_battery(+battery_level)).
modeb(high_battery(+battery_level)).
modeb(not high_battery(+battery_level)).
modeb(screen_on(+screen_brightness)).
modeb(not screen_on(+screen_brightness)).
modeb(is_charging(+battery_charging)).
modeb(not is_charging(+battery_charging)).
modeb(morning(+time)).
modeb(not morning(+time)).
modeb(afternoon(+time)).
modeb(not afternoon(+time)).
modeb(evening(+time)).
modeb(not evening(+time)).
```

We were able to solve the task under both optimal and general solution settings much faster than PASPAL, while ASPAL ran out of memory and crashed (see table 1).

All runtimes were measured by means of the `time` bash command according to the *real* amount of time spent upon calling the respective python script on *corona* lab machines (for full details on their hardware, see [11]). For our Parallelizer and PASPAL, this includes the time taken for preprocessing the given ILP task and splitting it into subtasks, calling ASPAL to solve each of the subtasks in parallel and postprocessing their solutions in order to obtain the solutions of the original task.

We have also noticed a problem in PASPAL. It places the two mode declarations with predicate *accept* into two separate subtasks, each including all

Table 1: Runtimes of Parallelizer, PASPAL and ASPAL on the original mobile task

	general solutions	optimal solutions
Parallelizer	22m34s	25m11s
PASPAL	51m36s	–
ASPAL	ran out of memory and crashed after 26m	ran out of memory and crashed after 30m50s

of the examples of the predicate. This way, the final solutions has to explain each example twice, which changes the semantics of the original ILP task. As a result, the subtasks with head declarations for the predicate *accept* crashed, whereas PASPAL output the product of the solutions for the rest of the subtasks instead of a solution for the original task, i.e. its output was incomplete and thus invalid. In our split, we have kept all head declarations and examples of a single predicate together.

After demonstrating the superiority of our approach within the problem domain solvable by PASPAL, we enriched the mode declarations so that more dependencies between head declaration predicates are present. We have also substituted each rule in the background knowledge with a head declaration and suitable examples for the predicate in the head of the rule, as well as body declarations for predicates corresponding to the ones found in the body of the rule. Again in the background knowledge, we have then reduced the range of terms that the type predicates may take so as to make computation feasible.

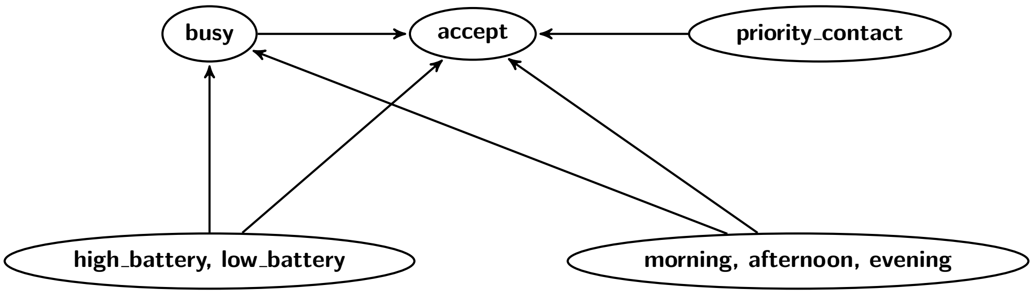
The rules we have removed from the background knowledge are shown below.

```
low_battery(X) :- battery_level(X), X <= 10.
high_battery(X) :- battery_level(X), X > 50.
date((D,M,Y)) :- year(Y), month(M), dayofmonth(D).
morning(H) :- time(H), H < 12, H >= 0.
afternoon(H) :- time(H), H >= 12, H < 18.
evening(H) :- time(H), H >= 18.
```

The mode declarations we have added are the following:

```
% head declarations
modeh(low_battery(+battery_level)).
modeh(high_battery(+battery_level)).
modeh(date((+dayofmonth,+month,+year))).
modeh(morning(+time)).
modeh(afternoon(+time)).
modeh(evening(+time)).

% body declarations
modeb(not busy(+date,+time,+volume,
  +vibrator,+battery_level,+screen_brightness,
  +light_level,+battery_charging)).
modeb(priority_contact(+contact)).
modeb(battery_level_less_or_equal_ten(+battery_level)).
modeb(battery_level_greater_than_fifty(+battery_level)).
modeb(time_less_than_twelve(+time)).
modeb(time_greater_or_equal_twelve(+time)).
modeb(time_less_than_eighteen(+time)).
modeb(time_greater_or_equal_eighteen(+time)).
```

The simplified reduced dependency graph of the enriched ILP task as constructed by our Parallelizer (remark 4.2.1) is shown below.



The full ASPAL programs are included in the final project archive. One general and one optimal solution for each of them are listed in appendix A. A comparison of the runtimes of our Parallelizer and ASPAL is to be found in table 2.

Table 2: Runtimes of Parallelizer and ASPAL on the enriched mobile task

	general solutions	optimal solutions
Parallelizer	10m17s	11m1s
ASPAL	ran out of memory and crashed after 17m34s	ran out of memory and crashed after 19m26s

Note that we have only measured the time PASPAL takes to compute the general solutions of the original ILP task as it is not able to compute optimal solutions. In case of the enriched ILP task, which contains connected head predicates, PASPAL passes the given ILP task to ASPAL without any changes in compliance with remark 6.1.4, so a separate measure of its runtime would not have made sense.

We have hereby shown that our Parallelizer is able to handle any ILP task solvable by PASPAL and not by ASPAL, but also tasks not solvable in either of the two frameworks.

The main advantages of our Parallelizer over PASPAL include the correct implementation of the split it employs, as well as the solid theoretical framework it is built upon. This enables it to split any given ILP task according to its reduced dependency graph even in the cases when PASPAL cannot produce a split and lets ASPAL solve the original task without amending it.

The main advantage of our Parallelizer over ASPAL is its capability to solve some large-scale ILP tasks intractable in ASPAL. This is due to the reduced mode bias of each of the subtasks compared to the given task, which may circumvent the memory overflow during grounding of the corresponding meta tasks. The huge difference a split makes is attributed to the grounding size potentially growing exponentially in respect to the size of the hypothesis space of an ILP task.

8 Conclusion and future work

The approach for splitting ILP tasks we have proposed in section 3 is novel and sound. It does also subsume all previous attempts at parallelizing ILP known to the author as summarized in section 6.4.

Its major setback is the incompleteness of the split within a connected component of the dependency graph under optimal solution settings (definition 3.2.4), although a non-empty subset of (slightly sub-) optimal solutions was produced in all tests we have conducted with ILP tasks, as well as in all examples but 3.2.6 in this report. A possible solution of this problem other than resorting to general solutions, as done in definition 3.2.7, is to introduce an iterative computation mode, as explained in section 8.2.

A split, by means of which each of the head mode declarations is handled with in a separate subtask, would be a further improvement of our approach. Such a split is feasible once we use abduction on the set of head mode declarations as elucidated in the following section.

8.1 Alternative approach to parallelizing ILP tasks

We have contemplated an alternative split of a given inductive learning task, which is more fine-grained but requires additional post-processing. We illustrate the underlying idea for a simple task with sole predicates p and q, such that p is dependent on q. We solve their subtasks, enriched with the set of abducibles $\{p, q\}$, in parallel, and then include in the solution set of the original task only these elements of the product of the two solutions, for which whenever q is abduced in the corresponding answer set of the subtask of p, it is abduced in the corresponding answer set of the subtask of q as well.

Theorem 8.1.1. For an inductive learning task $< B, E, M >$, let $Abd = pred(M_h)$, $n = |Abd|$ and $abd = 0\{a \mid a \in Abd\}n$. be the count aggregate for Abd. Then we have
$\bigcup\{\{r \mid r \in S_{M_i} \wedge meta(r) \in as_i\} \mid 1 \leq i \leq n\} \in ASPAL(< B, E, M >) \Leftrightarrow$
$\forall 1 \leq i \leq n\ \forall a \in Abd \cap as_i\ a \in as_a$,

where each as_i denotes an answer set of the abductive meta task corresponding to the ILP task $< B \cup abd, E_{\{p_i\}}, < mode(p_i), M_b >>$ and each S_{M_i} – the skeleton rules of the same ILP task.

In order to facilitate efficiency, we can formulate a hybrid approach, in which we construct the dependency graph of a given inductive learning task first (definition 3.1.3) and then apply the split described in 8.1.1 within each of its connected components rather than on the whole set of predicates.

Theorem 8.1.2. For an inductive learning task $< B, E, M >$ and a connected component $CC \in connComp(depGr(B, M))$, let $Abd = pred(M_{CC})$, $n = |Abd|$ and $abd = 0\{a \mid a \in Abd\}n$. be the count aggregate for Abd. Then we have
$\bigcup\{\{r \mid r \in S_{M_i} \wedge meta(r) \in as_i\} \mid 1 \leq i \leq n\} \in$
$ASPAL(< B, E_{CC}, < M_{CC}, M_b >>) \Leftrightarrow$
$\forall 1 \leq i \leq n\ \forall a \in Abd \cap as_i\ a \in as_a$,

where each as_i denotes an answer set of the abductive meta task corresponding to the ILP task $< B \cup abd, E_{\{p_i\}}, < mode(p_i), M_b >>$ and each S_{M_i} – the skeleton rules of the same ILP task.

Due to time constraints on the project, we were not able to implement or prove the correctness of either the alternative or the hybrid approach, notwithstanding their potential value.

8.2 Iterative computation mode

A further improvement of our split within each of the connected components (definitions 3.2.4 and 3.2.7), as well as the alternative and hybrid approaches from the previous section, is to consider batches of increasingly sub-optimal solutions for each subtask until a solution of the original task is found instead of considering exclusively optimal or general ASPAL solutions throughout. This is eventually going to provide us with a solution whenever one is present and thus outdo the incompleteness of these methods under optimal solution settings. It is also expected to do so faster than its counterpart dealing exclusively with general solutions and thus make computation feasible in practise.

8.3 Hybrid approach incorporating RASPAL

Another possibility for improving our approach is to call RASPAL in order to obtain the solutions of the subtasks we construct. RASPAL has been proven to be sound and complete in respect to both general and optimal ASPAL solutions (see [10]), which makes such a hybrid solution justifiable. It can also improve the runtime of RASPAL on its own, as well as the scalability of our Parallelizer in case any of the subtasks is big enough to make ASPAL run out of memory and crash. A big advantage of this hybrid approach is that it does not imply any changes to either of the systems it is combining.

References

[1] J. W. Lloyd. *Foundations of Logic Programming.* Springer-Verlag New York, Inc., 1984.

[2] Stephen Muggleton and Luc De Raedt. Inductive logic programming: Theory and methods, 1994. Available at `http://www.sciencedirect.com/science/article/pii/0743106694900353`.

[3] Marek Sergot. Minimal models and fixpoint semantics for definite logic programs, 2005. Lecture notes of the Knowledge Representation course given at Imperial College London, available at `https://www.doc.ic.ac.uk/~mjs/teaching/KnowledgeRep491/Fixpoint_Definite_491-2x1.pdf`.

[4] Vladimir Lifschitz and Hudson Turner. Splitting a logic program. In *Proceedings of the Eleventh International Conference on Logic Programming*, pages 23–37. MIT Press, 1994. Available at `http://dl.acm.org/citation.cfm?id=189883.189894`.

[5] Martin Gebser, Roland Kaminski, Benjamin Kaufmann, Marius Lindauer, Max Ostrowski, Javier Romero, Torsten Schaub and Sven Thiele. *A User's Guide to gringo, clasp, clingo, and iclingo*, 2010. Available at `https://www.cs.utexas.edu/users/vl/teaching/lbai/clingo_guide.pdf`.

[6] Domenico Corapi, Alessandra Russo and Emil Lupu. Inductive logic programming in answer set programming. In *Proceedings of the 21st International Conference on Inductive Logic Programming*, pages 91–97. Springer-Verlag, 2012. Available at `http://dx.doi.org/10.1007/978-3-642-31951-8_12`.

[7] Domenico Corapi, Alessandra Russo and Emil Lupu. Inductive logic programming as abductive search. In *Technical Communications of the International Conference on Logic Programming*, pages 53–63, 2010. Available at `http://wp.doc.ic.ac.uk/arusso/wp-content/uploads/sites/47/2015/01/TAL.pdf`.

[8] Mark Law. Non monotonic logic-based learning, 2015. Summary of units 6-9 of the Logic-based Learning course (C304) given at Imperial College London, available at `https://www.doc.ic.ac.uk/~ml1909/teaching/Non-monotonic%20Logic-based%20Learning,%20Summary.pdf`.

[9] Timothy van Bremen. PASPAL: A parallel logic-based learning system. Undergraduate Project Final report, supervised by Alessandra Russo and Mark Law, Imperial College London, 2015.

[10] Duangtida Athakravi, Domenico Corapi , Krysia Broda and Alessandra Russo. Learning Through Hypothesis Refinement Using Answer Set Programming. In *Inductive Logic Programming: 23rd International Conference, ILP 2013, Rio de Janeiro, Brazil, August 28-30, 2013, Revised Selected Papers*, pages 31–46. Springer Berlin Heidelberg, 2014. Available at `http://dx.doi.org/10.1007/978-3-662-44923-3_3`.

[11] DoC Computing Support Group. Teaching labs: Workstations. Available at `https://www.doc.ic.ac.uk/csg/facilities/lab/workstations`.

Appendix A Solutions of the original and the enriched ILP task on the mobile dataset

A.1 A general solution of the original ILP task on the mobile dataset

```
accept(A,-1,B,C,D,E,F,G):-date(A),volume(B),vibrator(C),
   battery_level(D),screen_brightness(E),light_level(F),
   battery_charging(G),not is_charging(G).

accept(A,1,B,C,D,E,F,G):-date(A),volume(B),vibrator(C),
   battery_level(D),screen_brightness(E),light_level(F),
   battery_charging(G),not is_charging(G).

accept(A,3,B,C,D,E,F,G):-date(A),volume(B),vibrator(C),
   battery_level(D),screen_brightness(E),light_level(F),
   battery_charging(G),not is_charging(G),screen_on(E).

accept(A,5,B,C,D,E,F,G):-date(A),volume(B),vibrator(C),
   battery_level(D),screen_brightness(E),light_level(F),
   battery_charging(G),not is_charging(G).

accept(A,7,B,C,D,E,F,G):-date(A),volume(B),vibrator(C),
   battery_level(D),screen_brightness(E),light_level(F),
   battery_charging(G),not is_charging(G).

accept(A,8,B,C,D,E,F,G):-date(A),volume(B),vibrator(C),
   battery_level(D),screen_brightness(E),
   light_level(F),battery_charging(G).

accept(A,5,B,C,D,E,F,G):-date(A),volume(B),vibrator(C),
   battery_level(D),screen_brightness(E),light_level(F),
   battery_charging(G),screen_on(E).

accept(A,7,B,C,D,E,F,G):-date(A),volume(B),vibrator(C),
   battery_level(D),screen_brightness(E),light_level(F),
   battery_charging(G),not low_battery(D).

accept(A,6,B,C,D,E,F,G):-date(A),volume(B),vibrator(C),
   battery_level(D),screen_brightness(E),light_level(F),
   battery_charging(G),not is_charging(G).

accept(A,3,B,C,D,E,F,G):-date(A),volume(B),vibrator(C),
   battery_level(D),screen_brightness(E),light_level(F),
   battery_charging(G),not low_battery(D),
   not is_charging(G).
```

```
accept(A,4,B,C,D,E,F,G):-date(A),volume(B),vibrator(C),
   battery_level(D),screen_brightness(E),light_level(F),
   battery_charging(G),not low_battery(D).

accept(A,-1,B,C,D,E,F,G):-date(A),volume(B),vibrator(C),
   battery_level(D),screen_brightness(E),light_level(F),
   battery_charging(G),not low_battery(D).

accept(A,4,B,C,D,E,F,G):-date(A),volume(B),vibrator(C),
   battery_level(D),screen_brightness(E),light_level(F),
   battery_charging(G),screen_on(E).

accept(A,0,B,C,D,E,F,G):-date(A),volume(B),vibrator(C),
   battery_level(D),screen_brightness(E),light_level(F),
   battery_charging(G).

accept(A,6,B,C,D,E,F,G):-date(A),volume(B),vibrator(C),
   battery_level(D),screen_brightness(E),light_level(F),
   battery_charging(G),not high_volume(B).

busy(A,B,C,D,E,F,G,H):-date(A),time(B),volume(C),
   vibrator(D),battery_level(E),screen_brightness(F),
   light_level(G),battery_charging(H).
```

A.2 An optimal solution of the original ILP task on the mobile dataset

```
accept(A,8,B,C,D,E,F,G):-date(A),volume(B),vibrator(C),
   battery_level(D),screen_brightness(E),
   light_level(F),battery_charging(G).

accept(A,3,B,C,D,E,F,G):-date(A),volume(B),vibrator(C),
   battery_level(D),screen_brightness(E),light_level(F),
   battery_charging(G),not is_charging(G),screen_on(E).

accept(A,4,B,C,D,E,F,G):-date(A),volume(B),vibrator(C),
   battery_level(D),screen_brightness(E),light_level(F),
   battery_charging(G),not low_battery(D).

busy(A,B,C,D,E,F,G,H):-date(A),time(B),volume(C),
   vibrator(D),battery_level(E),screen_brightness(F),
   light_level(G),battery_charging(H).
```

A.3 A general solution of the enriched ILP task on the mobile dataset

```
accept(A,6,B,C,D,E,F,G):-date(A),volume(B),vibrator(C),
   battery_level(D),screen_brightness(E),light_level(F),
   battery_charging(G),not low_battery(D).

accept(A,5,B,C,D,E,F,G):-date(A),volume(B),vibrator(C),
   battery_level(D),screen_brightness(E),light_level(F),
   battery_charging(G),not low_battery(D).

accept(A,4,B,C,D,E,F,G):-date(A),volume(B),vibrator(C),
   battery_level(D),screen_brightness(E),light_level(F),
   battery_charging(G),screen_on(E).

accept(A,6,B,C,D,E,F,G):-date(A),volume(B),vibrator(C),
   battery_level(D),screen_brightness(E),light_level(F),
   battery_charging(G),not is_charging(G).

accept(A,8,B,C,D,E,F,G):-date(A),volume(B),vibrator(C),
   battery_level(D),screen_brightness(E),light_level(F),
   battery_charging(G).

accept(A,7,B,C,D,E,F,G):-date(A),volume(B),vibrator(C),
   battery_level(D),screen_brightness(E),light_level(F),
   battery_charging(G),not is_charging(G).

accept(A,0,B,C,D,E,F,G):-date(A),volume(B),vibrator(C),
   battery_level(D),screen_brightness(E),light_level(F),
   battery_charging(G).

accept(A,5,B,C,D,E,F,G):-date(A),volume(B),vibrator(C),
   battery_level(D),screen_brightness(E),light_level(F),
   battery_charging(G),not is_charging(G).

accept(A,4,B,C,D,E,F,G):-date(A),volume(B),vibrator(C),
   battery_level(D),screen_brightness(E),light_level(F),
   battery_charging(G),not low_battery(D).

accept(A,7,B,C,D,E,F,G):-date(A),volume(B),vibrator(C),
   battery_level(D),screen_brightness(E),light_level(F),
   battery_charging(G),not low_battery(D).

accept(A,3,B,C,D,E,F,G):-date(A),volume(B),vibrator(C),
   battery_level(D),screen_brightness(E),light_level(F),
   battery_charging(G),not is_charging(G),screen_on(E).

accept(A,1,B,C,D,E,F,G):-date(A),volume(B),vibrator(C),
   battery_level(D),screen_brightness(E),light_level(F),
   battery_charging(G),not is_charging(G).

accept(A,3,B,C,D,E,F,G):-date(A),volume(B),vibrator(C),
   battery_level(D),screen_brightness(E),light_level(F),
   battery_charging(G),not low_battery(D).

accept(A,5,B,C,D,E,F,G):-date(A),volume(B),vibrator(C),
   battery_level(D),screen_brightness(E),light_level(F),
   battery_charging(G),screen_on(E).

accept(A,-1,B,C,D,E,F,G):-date(A),volume(B),vibrator(C),
   battery_level(D),screen_brightness(E),light_level(F),
   battery_charging(G),not is_charging(G).

busy(A,B,C,D,E,F,G,H):-date(A),time(B),volume(C),
   vibrator(D),battery_level(E),screen_brightness(F),
   light_level(G),battery_charging(H).

date((A,B,C)):-dayofmonth(A),month(B),year(C).

low_battery(A):-battery_level(A),not high_battery(A).

high_battery(A):-battery_level(A),not low_battery(A).

evening(A):-time(A),time_greater_or_equal_eighteen(A).

morning(A):-time(A),time_less_than_twelve(A).

afternoon(A):-time(A),not morning(A),not evening(A).
```

A.4 An optimal solution of the enriched ILP task on the mobile dataset

```
accept(A,3,B,C,D,E,F,G):-date(A),volume(B),vibrator(C),
   battery_level(D),screen_brightness(E),light_level(F),
   battery_charging(G),not is_charging(G),screen_on(E).

accept(A,B,C,D,E,F,G,H):-date(A),contact(B),volume(C),
   vibrator(D),battery_level(E),screen_brightness(F),
   light_level(G),battery_charging(H),not low_battery(E).

busy(A,B,C,D,E,F,G,H):-date(A),time(B),volume(C),
   vibrator(D),battery_level(E),screen_brightness(F),
   light_level(G),battery_charging(H).

date((A,B,C)):-dayofmonth(A),month(B),year(C).

high_battery(A):-battery_level(A),not low_battery(A).

low_battery(A):-battery_level(A),not high_battery(A).

evening(A):-time(A),time_greater_or_equal_eighteen(A).

afternoon(A):-time(A),time_greater_or_equal_twelve(A).

morning(A):-time(A),time_less_than_twelve(A).
```