MEng Individual Project

Imperial College London

Department of Computing

Learning Automobile Architectural Descriptions

Author:
Cristian Chirac

Supervisor:
Dr. Krysia Broda

Second Marker:
Dr. Alessandra Russo

October 23, 2018

Abstract

Automobiles nowadays, and particularly hybrid cars, are becoming more and more complex; they span multiple and various types of components, from engines to batteries, from gearsets to torque couplers and axles, and so on. On top of that, another level of complexity is added by all the ways in which these components can be connected to each other, and we can end up with millions of different types of architectures from only a dozen of components. Moreover, each architecture has its own particularities, such as certain patterns in which the components are interconnected, certain components that must or must not be connected, and so on. These patterns are usually non-trivial, so they cannot be expressed with ease in order to classify the architectures using them. As a result, a person is left with the task to look at every single architecture, represented as a typed graph, analyze it for several seconds or even minutes, and then decide on its class. Having to do this for each of, say, a million cars can end up taking a very long time.

This project, as its title suggests, is trying to take that burden off the user and instead learn the rules the engineer uses when classifying architectures and then perform the classification automatically for the entire dataset. We first look at the best ideas and concepts to use in order to achieve this, the main focus being on *answer sets* and in particular a tool called ILASP, then we steadily build up to a learning process that achieves perfect accuracy on a given dataset. We then integrate this learning process (using ILASP) into an interactive cycle we shall call *classilasp*, where the user is repeatedly given models and asked to classify them, then *classilasp* does the rest of the job, incrementally learning the rules it needs from those models. Finally, once its basic functionality is covered, we look at the most important features and optimizations we have built on top of that in order to make *classilasp* as quick, reliable, interactive and user-friendly as possible.

In the end, after all these are presented and explained theoretically, we present practical evaluations, both quantitative and qualitative, and then look at some potential extensions that could be integrated in the future in order to further improve *classilasp*'s capabilities.

Acknowledgements

I would like to thank my supervisor and tutor, Dr. Krysia Broda, for her incredible dedication, contagious cheerfulness and willingness to help me whenever I needed it throughout these four great years.

I would also like to thank the entire Siemens team, and in particular Dr. Joana Côrte-Real, Dr. Cameron Sobie, Dr. Mike Nicolai and Dr. Johan Vanhuyse, for proposing the project and providing the supporting data as well as for all their help, feedback and very useful and interesting suggestions

Given that this project is centered around the learning tool ILASP, I must also thank its author, Mark Law, as well as for his input and help.

Additionally, I send my gratitude to my former Maths, Computer Science and Physics teachers, Mr. Romeo Zamfir, Ms. Mirela Voinea, Ms. Adina Birsan and Mr. Mihai Vasiliu; they have always inspired and motivated me throughout the years, both academically and personally.

Finally, it goes without saying that I wouldn't be the person I am today without my family and my closest friends, I owe them everything and love them dearly.

Contents

Chapter 1

Introduction

1.1 Motivation

This project was proposed by Dr. Mike Nicolai and Dr. Cameron Sobie from Siemens AG, which is "a German conglomerate company headquartered in Berlin and Munich and the largest industrial manufacturing company in Europe with branch offices abroad" [9], one of which is where they currently work, located near the city of Leuven in Belgium, and is mainly focused on research.

The project is based on quite a simple problem, in principle, with no clear, obvious solution though. Take, for example, the very first specific dataset that was presented to us: hybrid cars. They are gaining more and more attention on the global scale, as new regulations are being established ([10] discusses this, specifically for the European Union), and as a result more and more different variations of such automobiles are being developed. This also means more ways in which they are constructed: they are not simply one engine powering the axles, but can instead contain multiple types of ever larger components. Moreover, the way in which these components are inter-connected is another defining aspect of an architecture.

Obviously, as the number of inner components grows linearly, the ways in which we can draw edges between them grows more or less exponentially, as this is a simple combinatorial problem. Clearly, this is a process that can be done automatically, and Siemens have indeed developed a tool to do just this. However, not all generated architectures are valid: for example, some components should never be connected to each other, either for a trivial reason that can be added to the combinatorial generation algorithm in order to not generate those in the first place, or for a more subtle reason that an engineer has to manually inspect and asses.

Note that, in this case, the engineer would simply inspect architectures in order to decide on their validity, in other words to *classify* them as either **good** or **bad**. However, these architectures can often have certain characteristics, like similar *patterns*, *substructures*, that the user can spot and classify them in more specific classes. In the case of hybrid cars, for example, the *type of hybridization* of a car is a kind of classification: they are either **parallel**, **series** or **series-parallel**. Another type would be based on the car's axle drive: **front**, **rear** or **both**. These, in fact, are independent classifications, so an architecture can be a tuple of any number of labels from several separate classifications.

However, it should be noted that not all classifications are equally complex. Deciding on the type of axle drive is trivial (just inspect which axles are used in the architecture), whereas deciding on the type of hybridization is not only more complicated, as it involves inspecting the ways in which several components are connected, but it even becomes debatable between engineers, as different engineers may use different (personal) reasonings when choosing the label.

Now the problem becomes rather obvious. Given the massive number of architectures generated from a relatively small number of components, it becomes virtually impossible for engineers to assess and make these decisions for every single one of them. For example, it takes engineers at Siemens around 10-20 seconds to classify simple architectures and even up to a full minute for the more complex ones in the *type of hybridization* case; when the number of generated architectures is measured in hundreds of thousands, which quite often is the case, it can take a very long time to do the entire classification just for one dataset.

One last thing to note is that, although the initial data that was provided was specifically aimed at hybrid architectures classification, this is clearly a more general problem concerning typed graphs classification and should be treated as generically as possible; that way, any solution that we come up with will not only solve this specific given problem, but also the more general underlying principle.

1.2 Objectives

Obviously the main objective of this project was to solve, even partially, the problem stated above. In other words, the main goal was to

- **automatically classify correctly an entire (possibly very large) set of graphs (corresponding to the generated architectures) only based on a much smaller set of architectures that have been manually inspected and classified by human experts**.

Since even deciding which of the generated architectures are valid or not is a type of classification and requires manual inspection of each architecture, being able to automatize that was the very first objective.

Once/if that worked well enough, we could then move on to do some of the following:

- do more complex classifications (e.g. type of hybridization instead of just valid/invalid).
- build an (online) tool that would generate random architectures and give them to engineers to classify, thus incrementally learning to classify the rest of the architectures until all of them are uniquely labeled based on the user's responses.
- using the capabilities of the learning tool that this project was centered around, namely ILASP, another suggestion was to generate something called "weak constraints", that would allow generating architectures more or less ordered by how "desirable" they are.
- as already stated above, it should remain one of the main objectives to never build our solution in a way specific to the particular problem we were given, and instead always come up with solutions that are as generic as possible, in order to also have applications in other fields.

1.3 Challenges

Naturally, the project involved quite a few challenges. In order to get used to what the graph representations of the architectures look like and to fully understand what the problem was, we were given by Siemens a set of architectures, and so the first challenge was to make sense of them.

In order to overcome this challenge, we decided to try to generate these architectures ourselves, which also helped with (re)visiting notions of Answer Set Programming, which were one of the core concepts behind this project, and also with learning the syntax for this type of programming, for tools such as *clingo* and ILASP.

Another challenge was that the project was quite open-ended; the problem was clear enough in principle, but the specifics of how to tackle it, what strategies, notions, tools, even programming languages, to use, those were details that were decided upon even several months after the project began. Because some of them were indeed very important, everyone involved had to bring their input before a decision could be made, and the Siemens team proved very enthusiastic and helpful in this sense.

As will become apparent in the next few chapters, finding solutions to the learning problem presented above became less and less of an issue, and instead the speed of converging to those solutions became a concern. Initially, some rather basic tasks took several hours to arrive at the (admittedly correct) results, so one of the main challenges throughout the project was to speed up these processes, either completely programatically, through improved algorithms and parallelization, or using some of the user knowledge as learning "bias".

Also, at least initially, it seemed quite difficult to come across research papers addressing this kind of problem that the project revolves around. There were some papers that seemed promising, but in the end for most of them, their ideas and solutions didn't seem necessarily relevant to our task. We did find some that were absolutely vital though, as will be seen not only in the Background section, but also throughout our implementation chapters, from the ILASP and *clingo* manuals and papers, to how clustering can be used, passive/active learning and so on (all of these will, of course, be covered throughout this report).

In terms of logistical challenges, the main one was always going to be communication, given that the project not only involved constant discussions with the project supervisor, but also getting in touch from time to time with the co-supervisor and with the research team from Siemens that proposed the project, located in a different country, in order to make sure that the project was moving in a direction everyone was happy with. Since email was not always the quickest or most satisfactory solution for either party, conference calls (spanning up to three different countries) as well as an on-site visit proved to be much more effective.

1.4 Achievements

Having seen the goals and challenges, in this section we list the achievements accomplished during the project:

- We built an **incremental learning tool**, called *classilasp*, that is an efficient and reliable solution to the broad problem proposed by Siemens.

  Specifically, it offers certain models to the user based on some internal criteria (to be discussed later on), then uses the user's labels for these models to eventually learn the correct classification rules. When exactly this happens is entirely up to the user (they see the rules and can assess how correct they are), at which point the entire dataset can be **automatically classified** with no further required user input.

- We **optimized the actual classification process** at the end, so that, for example, instead of waiting several hours, or even days, to classify over 170.000 models, it took less than 2 minutes. This is discussed in detail in sections 6.5 and 7.1.2.

- We developed **several different strategies** when choosing what model to show to the user, so that we increase the probability to learn something "new" with every new classified model and converge to the right solution as quickly as possible. These include **continuous internal recalibration** of the algorithms as well as making full **use of the user's bias**.

- Finally, *classilasp* works with **generic input data** and is in no way dependent on it being one of the datasets we've worked with, thus following one of the main goals stated above.

Chapter 2

Overview

In this chapter we have a closer look at the initial data the Siemens team gave us, as well as the problem it came with, then we show a high-level overview of how we approached this problem before delving into the details in the chapters to follow.

2.1 Sample automobile architectures

Initially, we received a JSON file describing the automobile architectures as graphs. This is a sample of a raw, unparsed JSON object describing one such architecture:

```
{
        "systemConnections": [{
                "targetPort": "p9108",
                "target": "n9105",
                "derivedFrom": "n181",
                "source": "n9094",
                "sourcePort": "p9096",
                "type": "systemConnection",
                "id": "sc9109",
                "name": "rear_axle_0_to_port_rotational_2_0"
        },
        ...],
        "nodes": [{
                "comment": "",
                "owners": [{
                        "owner": "n9094",
                        "id": "n9092"
                }],
                "hasBlocks": "False",
                "name": "ICE_0",
                "alwaysReachable": "False",
                "derivedFrom": "n23",
                "multiplicity": "[1]",
                "mutableInterface": "undefined",
                "ports": [{
                        "comment": "",
                        "direction": "inout",
                        "group": "-1",
                        "name": "port_rotational_2_0",
                        "derivedFrom": "n24",
                        "location": "right",
                        "owner": "n9094",
                        "position": "0.5",
                        "type": "rotational",
                        "id": "p9096"
                }],
                "allowEmpty": "False",
                "color": {
                        "r": "255",
                        "b": "80",
                        "g": "80"
                },
                "allowSelfConnections": "none",
                "forceInsideConnections": "False",
                "attributes": [],
                "blockType": "PhysicalBlock",
                "type": "Component",
                "id": "n9094",
                "partOf": "9092"
        },
        ...],
        "type": "Architecture",
        "id": "9092"
}
```

Obviously, a lot of data was skipped from the object above because of its overall textual size - it remains large enough as it is, considering it only describes one node and one edge of the graph! However, it's quite clear what the general graph structure is and it already can be analyzed by a human being.

It's also obvious that there is **a lot** of redundant data that does not describe the architecture in any constructive way (e.g. comments, color, attributes) and simply adds to the unreadability of the graph representation, so we can parse all of these out and leave only the essential bits.

This can be done easily with a script, and we get a much simpler, more compact and more readable graph representation that fully describes an architecture. This is still quite large and we only list part of it below as proof of concept:

```
{
  "nodes": [
    {
      "id": "n9101",
      "name": "EM_0",
      "ports": [
        {
          "id": "p9103",
          "name": "port_electrical_9_0",
          "type": "electrical",
          "owner": "n9101"
        },
        ...
      ]
    },
    ...
  ],
  "edges": [
    {
      "source": "n9094",
      "sourcePort": "p9096",
      "target": "n9105",
      "targetPort": "p9108"
    },
    ...
  ]
}
```

A few things to mention here:

1. Although the **nodes** of the graph are described as the automobile components (EM - electric motor, ICE - Internal Combustion Engine, Battery, etc.), the edges are not drawn from them directly, but rather from the **ports** inside these components. This adds to the level of complexity of the graphs, but it's the more realistic approach, since these are the actual connections we see in real-life architectures.

   It should be noted that, as a result, the overall complexity of the generation problem is not really a function of the number of components, but the number of ports instead.

2. In some cases, the port we choose to use from a component when "drawing" an edge does make a difference, in that it generates a different, new architecture from the ones generated when using the other ports of that component. For example, the front and rear axles, regarded as ports on the same "vehicle" component, give the type of axle drive of the car and so they must generate **different** architectures.

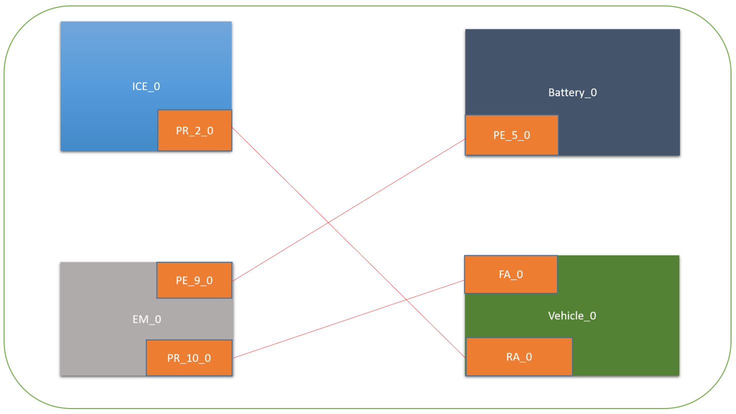
   Sometimes it doesn't matter and using different ports of the same component generates architectures that are **isomorphic** to each other (informally, there are some components with a list of *identical* ports and, when drawing the edges, we can choose "whichever port we want"; any choice generates an architecture of the same isomorphic class. As emphasized, this clearly doesn't apply to the front vs rear axle ports). We will revisit this notion a bit later on and see some proper examples, in case it is not clear enough yet.

   Thus we want to ideally generate only one representative for each class of isomorphic architectures. Choosing which components should have what type of ports is a design decision made by the engineer before providing it all to the generation tool, as he is the only one who knows and can assess which option is best.

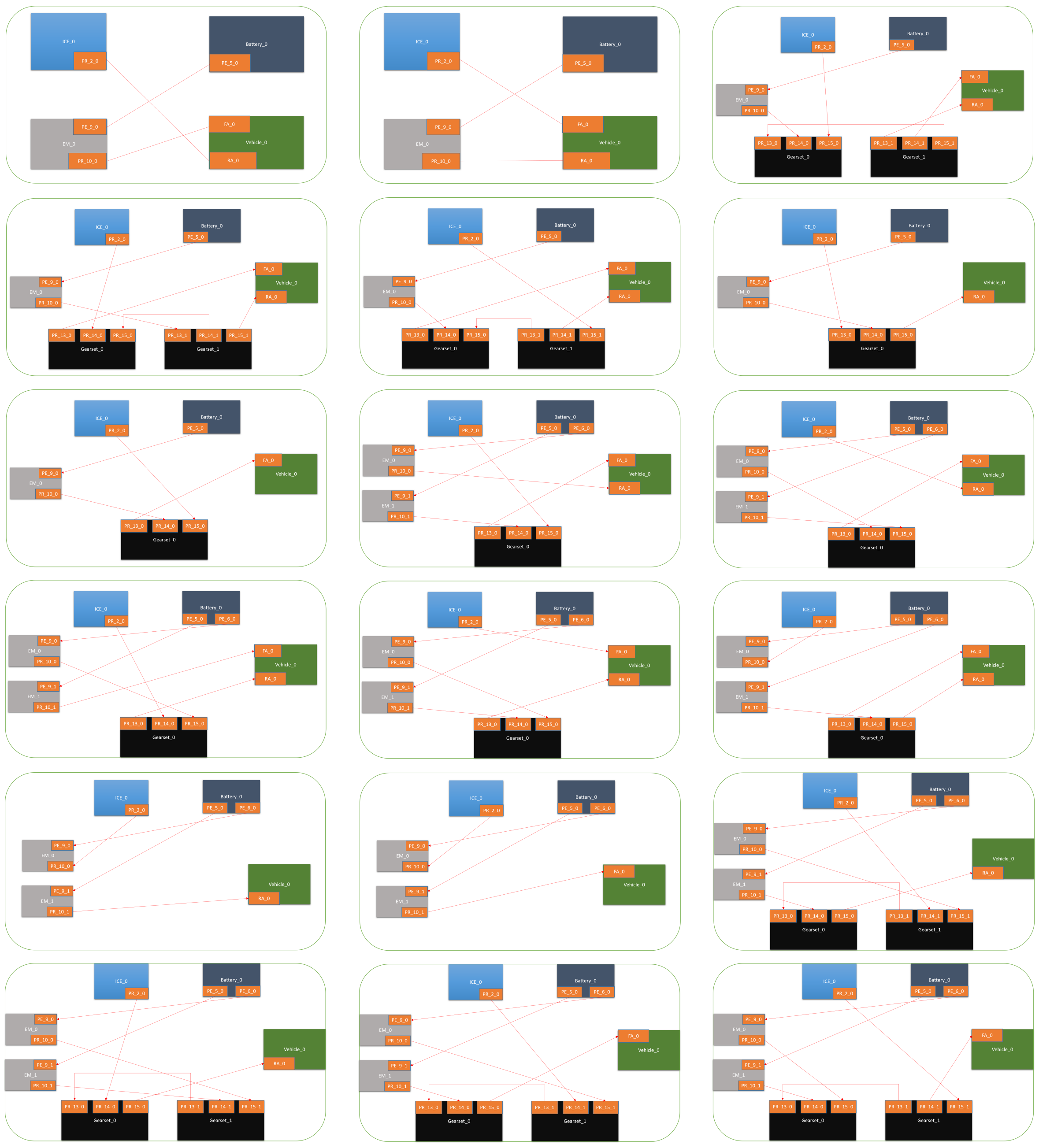
3. Even though the edges are described as going *from* a sourcePort *to* a targetPort, we don't regard them as being directed, since the direction of the edges only really matters on a physical level (architecturally they really are simple, undirected connections between two ports). This is also safer when trying to avoid generating identical architectures (we don't want the simple action of inverting an edge to generate a different architecture, so we avoid this risk by removing direction altogether). As we shall see, this is also vital for the kinds of "patterns" we can learn in these graphs.

This object can now be used to draw a simple diagram describing the corresponding architecture, which makes it easier to visualize. What the best way to draw these is also a good question. Initially, we didn't know of a good tool to use for this, given that these are not *normal* graphs with "atomic" nodes, but instead graphs with nodes *within* higher-level nodes (i.e. ports inside components). As a result, knowing that there were only 18 models in totals, the diagrams presented below were constructed manually, using PowerPoint drawing tools, and only later on we automatized this process.

For example, the graph corresponding to the object partially presented above was:



However, when we list all 18 architecture diagrams, we can already make some high-level observations and deductions from them based on which components are used, how they are connected, which ports are interconnected, and so on:



It’s interesting to see how each component and port can (or cannot) connect to others, allowing us to understand the kinds of **constraints** that were used for generating them without actually knowing them.

Only now is it really suitable for an expert to look at these diagrams and promptly decide which classes each of them belongs to, be it type of hybridization, axle drive, and so on. In fact, what we did here is exactly the process that the design tool from Siemens goes through, including the diagrams.

Note that these 18 architectures were generated with the following components:

- **vehicle** (exactly one) - with 2 rotational ports: front axle, rear axle
- **ICE** (exactly one) - with 1 rotational port
- **battery** (exactly one) - with 2 electrical ports
- **EM** (between one and two) - with 1 electrical port and 1 rotational port
- **gearset** (between none and two) - with 3 rotational ports

There are some other constraints that the tool uses when generating valid architectures (some ports are optional - they may or may not be used in an architecture, some aren’t; also, you can only connect ports of the same type), but it’s important to note now what was emphasized earlier on: **only** 18 architectures were generated with these given components, even though we have 15 ports in total. There are, of course, the mentioned restrictions for how they can be connected, but for a graph with this many nodes we would surely expect more than 18 possibilities.

This is where we see how important it is to filter out architectures that are equivalent in functionality. We can observe, for example, that when both gearsets are used, they are only connected to each other via their port 13 (the leftmost one in the diagrams). This is because if we connected them via (say) port 14, then we can swap in both of them whatever is connected to ports 13 and 14 and the overall architecture would behave the same way. This is because of something called **port groups**, which formally define what we previously informally described as the ability to "choose whichever port we want" from that group.

If two or more ports of the same component are defined to be part of the same port group, then connecting an external port to either of them makes no isomorphic difference. Specifically, gearsets contain 3 ports (with ids 13, 14, 15), all belonging to the same port group. This allows us to do the "swapping" of connections from before and still obtain isomorphic architectures.

When generating the valid architectures without considering port groups, we get a total number of 1198, instead of just 18. Given that, even after filtering out isomorphic architectures, the Siemens tool generated as many as 700.000 architectures for some configurations, the importance of port groups becomes obvious.

So it’s easy to see now how this dataset is very useful, as it allows us to understand some important aspects, such as what the problem is and why it needs to become a more automated process, but it also gives us enough information to see how we could generate valid architectures ourselves. Siemens is currently using a SAT solver tool that has some hard, permanent constraints (e.g. never connect a rotational port to an electrical port), as well as custom constraints (e.g. what components to use, what ports are optional, what the port groups are) that are supplied by an engineer. Instead, we used these given architectures to deduce most of the intuitive constraints and general rules that make up the generative model, and used Answer Set Programming and namely *clingo* to construct them ourselves.

2.2 Learning class rules

An important thing was that we didn't actually know the classification rules for the hybrids case ourselves, so that we wouldn't bias the learning process, and instead had only the pre-computed classifications, done by Siemens. So we investigated what kinds of such rules we could **automatically** learn for these models, using ILASP.

And in fact, the results were more than satisfactory: clear, concise hypotheses were outputted, which produced 100% accuracy when used for classifying the entire dataset (admittedly, of very small size). Take the following sample hypothesis for the *parallel* class:

```
invented_pred :- comp(V1, battery), direct_path(V0, V1, V2).
parallel :- not invented_pred.
parallel :- comp(V1, vehicle), direct_path(V0, V1, V2).
```

Even though its structure and predicates are explained later on, we see how quite a simple classification hypothesis is learned and only dependent on two components: the **battery** and the **vehicle**. This means it is not over-fitted to the data, and instead learns a more general pattern specific to the **parallel** class (that pattern is described by the **direct_path** predicate, as will be formally defined).

This convinced us that an interactive tool as we intended could indeed be possible. Clearly, whatever manual learning process we did here would be done automatically by this tool, but if we couldn't achieve satisfactory results ourselves, the tool would clearly not be able to either.

2.3 Learning cycle - *classilasp*

The learning cycle tool we built, *classilasp*, has the following "high-level" steps:

1. parse the input file in order to "make sense" of it for the following steps
2. choose models from the input file and display them to the user, recording the labels they select for them
3. use those models (with their labels) to automatically construct the ILASP programs in order to learn the corresponding class hypotheses
4. when the user decides the current class hypotheses are correct, use those to automatically classify all the models from the input dataset

Note that the first step is virtually a **pre-processing** stage, the last - **post-processing**, while the other two describe the actual interactive learning cycle between the user and the tool. This can be visualized with the following diagram:

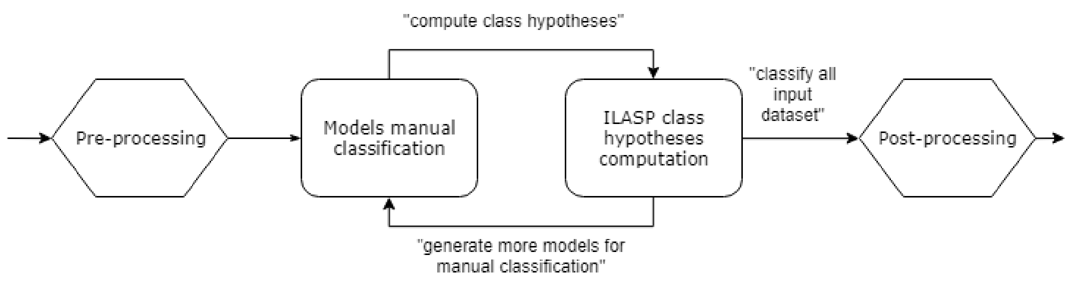

"compute class hypotheses"
Pre-processing
Models manual classification
ILASP class hypotheses computation
"classify all input dataset"
Post-processing
"generate more models for manual classification"

Figure 2.1: High-level learning tool structure

Note that the labels in quotation marks are user commands. This is obviously a very general overview of how *classilasp* works, the details of which are discussed in much more detail in chapter 5. It's also worth noting that all these parts are completely independent in implementation and are, thus, modularized as much as possible. In particular, the tool should not make any specific assumptions about how the ILASP process(es) run, and instead only know what kind of input to provide and what kind of output to expect.

2.4 Further features and optimizations

After discussing the basis of the learning tool, we show how several powerful features and optimizations were built on top of it. They tackle issues in all four main steps presented above, in terms of functionality as well as usability, and all of chapter 6 is dedicated to presenting and analyzing them.

The most important sections discussed there include:

- **continuous recalibration** of the algorithm choosing the models for manual classification, based on the latest computed class hypotheses
- **prioritization** of certain models for manual classification, based on their probabilities to improve the class hypotheses
- defining various graph **patterns/substructures** that can be relevant for the class hypotheses in the learning process
- **querying** the input dataset using user-written constraints, in order to obtain models that they consider important/relevant for classification
- **removing multiprocessing overhead** in order to greatly improve computation time for automatic classification as well as queries
- ability to compute **accurate class hypotheses** even with (slightly) **inaccurate manually classified data**

2.5 Evaluation and Conclusion

Finally, we evaluate the project both quantitatively and qualitatively. The former takes some of the most important global variables in *classilasp* and analyzes how some vital aspects, such as computation time or used resources, vary with their values. For example, since a lot of parallelization is used in order to speed up the overall task, we look at how more parallelization can improve things, and how much is too much.

classilasp, through ILASP's capabilities, can also deal with inaccurate manual classifications from the user, up to a certain point. We also, thus, analyze what that "certain point" is and how the accuracy of the learned class hypotheses depends on the accuracy of the user data.

Since quite a lot of user interaction is present in *classilasp* in the end, we analyze how smooth and intuitive it generally is, as well as how testable and extensible the application is, in the qualitative evaluation section, before presenting the future work and potential extensions in the Conclusion chapter.

Finally, in order to download and use *classilasp*, please consult the **User guide** at the end, which comes with instructions and a sample run with screenshots in the most important steps involving user interaction.

Chapter 3

Background

In this chapter we introduce all theoretical notions needed in the remaining chapters.

3.1 SAT

The architecture generation tool from Siemens makes use of something called **The Boolean Satisfiability Problem** (SAT). For this section we extract some definitions and other basic notions from [28]. The SAT problem can be stated as follows:

- Given a boolean formula $F(x_1, ..., x_n)$, is it satisfiable? If yes, output configuration $x_1, ..., x_n$ that makes it true.

Obviously, there are 2^n different such configurations that could potentially make F true. This problem is clearly in NP, and in fact it was the first problem proven to be NP-complete. In general, F is represented in Conjunctive Normal Form (CNF), meaning that it is the conjunction of several disjunctive logical formulas using $x_1, ..., x_n$.

For example, let F be:

$$(x_1 \vee x_2 \vee x_3) \wedge (\neg x_1 \vee \neg x_2 \vee x_3) \wedge (x_1 \vee \neg x_2 \vee \neg x_3) \wedge (\neg x_1 \vee x_2 \vee \neg x_3)$$

It's quite easy to see that choosing all of $x_1,\ x_2$ and x_3 to be true makes F true, so it is indeed satisfiable. In general, though, it is not at all this easy to reason about non-symmetrical, complex formulas with hundreds or thousands of atoms. Thankfully, there are some aspects of CNF that certain algorithms can exploit and speed up the search process for a satisfying assignment (DP and especially DLL/DPLL [38], which is currently the one mostly used by solvers).

Because of the fact that SAT not only tells us whether the given formula is satisfiable or not, but also *generates* one (or more) satisfying solutions, it becomes clear why it can be such a powerful tool and it also suggests how it might be used by Siemens for generating their architectures.

First they have to define their predicates in the context of their components, ports, connections, etc. Then they have to create certain constraints with them, then join them together in a large conjunction. If not already in CNF, this logical formula needs to be brought to this form, and only then we can give this to a SAT solver that solves it and supplies us with a solution. This solution will tell us what the truth values of the original predicates (in terms of the structure of the architecture) are and thus provide a full definition of an architecture.

Finally, this can then be decoded into either a more human-readable form or, as Siemens does, generate diagrams with the graph representations, which makes it easier for engineers and designers to analyze.

3.2 Answer Set Programming

Closely related to the SAT solving approach is Answer Set Programming. Most of this section is inspired by [22], [13], [27], [34] and [21], which should be consulted for more details and further examples and applications.

Answer Set Programming (ASP) is a declarative paradigm used for solving complex search problems in the field of Knowledge Representation and Reasoning, mainly because of its great solving capabilities, yet rather simple, intuitive language. Basically, someone who is given a computational problem simply has to translate it into a logic program and then use an ASP solver to find all its valid solutions (called **answer sets**). What is remarkable about ASP is that it manages to solve all search problems in NP (and NP^{NP}) in a uniform way with arguably more intuitive problem representations than SAT [22], which is why it is a good candidate for our architecture generating problem (since it clearly has a complexity that's exponential in the number of available ports, but can be solved in NP by "guessing" a configuration for the graph representation of the architecture and then check in polynomial time that it doesn't break any given constraints).

In order to formally define ASP in the following pages, we will mostly use notions as formally defined in [13], and then refer to them either in this form or a more informal one throughout the rest of the report:

3.2.1 Definitions

An ASP **rule** has the following form:

$$p_0 \leftarrow p_1, ..., p_m,\ not\ p_{m+1}, ...,\ not\ p_n, \tag{3.1}$$

where $n \geq m \geq 0$, and we call each of $p_i,\ 0 \leq i \leq n$, **atom**s. Let r denote the rule in (3.1). Then:

- $head(r) = p_0$ is the head of the rule;
- $body(r) = \{p_1, ..., p_m,\ not\ p_{m+1}, ...,\ not\ p_n\}$ is the body of the rule;
- $body^+(r) = \{p_1, ..., p_m\}$;
- $body^-(r) = \{not\ p_{m+1}, ...,\ not\ p_n\}$;

A rule without a body is called a **fact**, and a rule without a head is called a (hard) **constraint**. The reason why the constraints are defined in this way is that an empty head defaults to *false*, so that the conjunction of all the atoms in the body of the rule entails falsity (so intuitively they should never all be *true* simultaneously).

A logic program P is **basic** if $body^-(r) = \emptyset,\ \forall r \in P$. A set of atoms X is **closed under** a basic program P if:

$$body^+(r) \subseteq X \rightarrow head(r) \in X,\ \forall r \in P.$$

We can now define $Cn(P)$ as the smallest set of atoms that is closed under a basic program P.

The **reduct** of a logic program P with respect to a set of atoms X (written P^X) is defined as:

$$P^X = \{head(r) \leftarrow body^+(r) \mid r \in P,\ body^-(r) \cap X = \emptyset\}$$

Intuitively, we can use the following two steps in order to obtain the reduct of P with respect to X, as shown in the example afterwards (adapted from [34]):

1. Remove from P all rules that contain in their body the negation as failure of an atom from X.

2. Remove any negation as failure atoms from the remaining rules of P.

This is one of the most classic, yet important examples in ASP:

Example. Let $X = \{p\}$ and the logic program P be comprised of the following two rules:

$$\begin{aligned} p &\leftarrow \mathit{not}\ q. \\ q &\leftarrow \mathit{not}\ p. \end{aligned}$$

Then, in order to obtain P^X, we apply the two steps from above. We can remove the second rule of P using step 1, since its body contains $\mathit{not}\ p$, and p is in X, and we can then remove $\mathit{not}\ q$ from the body of the first rule using step 2. This means that P^X consists of only one rule, which is the fact p.

One thing to note here, using the definitions from above, is that $X = Cn(P^X)$ and $P^X = X$, so in a sense we can already intuitively assess that X is somewhat "stable" in terms of what we can learn from program P based on it.

Using this observation, we can generalize this "stability" concept and define the **answer sets** of a program P to be those sets of atoms X for which $Cn(P^X) = X$ (they are, in fact, also called *stable models*). We can now go back to the example above and try to find all the answer sets of P. Obviously, as already noted, one of them is $\{p\}$, and since P is symmetrical in p and q, so is $\{q\}$. Given how $Cn(P)$ is defined, it's obvious that X should only contain atoms derivable from P, so the only two other candidate answer sets are $\emptyset$ and $\{p,\ q\}$.

However, when $X = \emptyset$, P^X becomes:

$$\begin{aligned} p. \\ q. \end{aligned}$$

so $Cn(P^X) = \{p,\ q\} \neq X$. Now, when $X = \{p,\ q\}$, both rules in P are removed in order to obtain P^X, which thus is $\emptyset \neq X$. So finally we conclude that neither of these two are answer sets of P, and its only answer sets are indeed just $\{p\}$ and $\{q\}$. What we've done here is we've basically generated the valid "solutions" to our problem (encoded into logic program P). Obviously, our program was very small and we only had to check 4 different candidate answer sets, but as the initial problem gets more complex, it gets encoded into an also more and more complex logic program. Thankfully, very efficient ASP solvers exist nowadays and we shortly discuss one (that ILASP, described later on, uses): *clingo*.

But before that, we need to introduce a few more concepts (all adapted from the manual at [27]) that are very important in ASP. Firstly, the **Herbrand base** of a program P is the set of all ground atoms that can be obtained using predicates, constants and functions from P. The **Herbrand Interpretations** (or simply "interpretations") of P are the subsets of its Herbrand Base, and they are called interpretations because they each assign a truth value to every atom in the Herbrand Base.

We also define **choice rule**s as:

$$l\ \{h_1; ...; h_m\}\ u \leftarrow b_1, ..., b_n.$$

having $0 \leq l \leq u \leq n$). It is a choice rule because between l and u atoms have to be true in an interpretation where $b_1, ..., b_n$ are also true for that interpretation to remain an answer set candidate (in other words, if this is not the case, then that interpretation cannot be an answer set). These, as we'll see later on, are a very powerful tool, both for generating solutions to a problem as well as restricting what those solutions might be.

Finally, note that, in general, there are programs with *no* answer sets as well as programs with *multiple* answer sets. However, we define a specific type of programs that, according to [19], have *exactly one* answer set: **stratified programs**. According to [35], a normal logic program P is stratified when it can be written as a disjoint partition $P = P_0 \cup P_1 \cup ... \cup P_n$ such that, for every predicate p, all its definitions (i.e. rules with p in their head) are in a *unique* P_i ($0 \leq i \leq n$) and for all k, $0 \leq k \leq n$, the following hold:

- if a predicate occurs positively in (i.e. non-negated in the body of) a rule in P_k, then its definitions are in P_j, with $j \leq k$
- if a predicate occurs negatively in (i.e. negated in the body of) a rule in P_k, then its definitions are in P_j, with $j < k$

3.2.2 gringo, clasp, clingo

The ASP solver we used for this project is called *clingo*. We give some details on how it works, how to use it, along with some examples, based on its official guide [21] and [22], along with some observations from [27] and [34]. Again, for more details and clarifications we strongly recommend you consult these sources.

The basic process of solving a logic program P is to first ground it, which obtains an equivalent program that doesn't contain any variables and only contains "ground" atoms instead. This is basically *gringo*'s job. Then the output of *gringo* (which is usually just numbers and is not normally human-readable) is fed into *clasp* as input, which is the actual solver that produces the answer sets.

As its name suggests, *clingo* is the combination of the two, so the following two UNIX calls are in fact equivalent [22]:

$ gringo myprogram | clasp
and
$ clingo myprogram

In other words, the following diagram describes the overall process of obtaining solutions to a problem using ASP (solving with *clingo*):

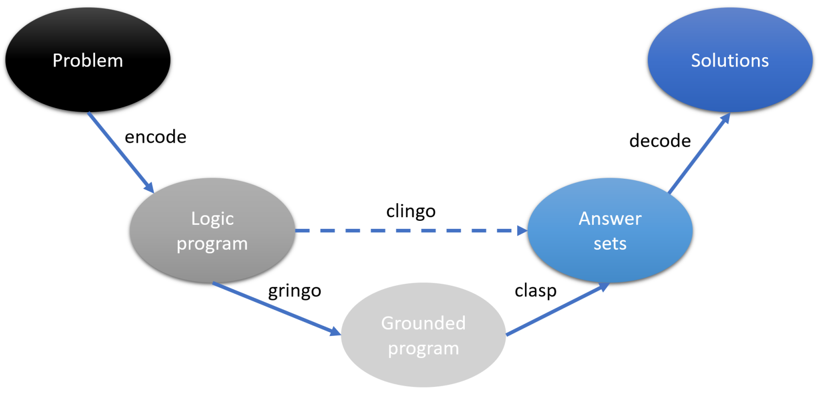


Figure 3.1: gringo, clasp, clingo

The syntax is pretty much the same as what we've described so far as ASP syntax, apart from the replacement of " $\leftarrow$ " with ":-" in rules. Other more in-depth syntax notions are either not relevant for this project or will be discussed in the ILASP section.

We can now consider a simple example adapted from [27]:

```
port(1). port(2).
1 { rotational(P); electrical(P) } 1 :- port(P).
```

This says that we have two ports, encoded as 1 and 2, and each can be exactly one of "rotational" or "electrical", which is a basic notion of the architectures we've seen so far. We would like to see what the possible outcomes might be. Running *clingo* with this problem generates the four possible solutions:

```
Solving...
Answer: 1
port(1) port(2) rotational(1) electrical(2)
Answer: 2
port(1) port(2) rotational(1) rotational(2)
Answer: 3
port(1) port(2) electrical(1) electrical(2)
Answer: 4
port(1) port(2) electrical(1) rotational(2)
SATISFIABLE

Models : 4
Calls : 1
Time : 0.037s (Solving: 0.04s 1st Model: 0.00s Unsat: 0.01s)
CPU Time : 0.000s
```

3.2.3 Generate-and-test

One of the most common approaches to writing an ASP program for a modeling problem is by using the "generate-and-test" method [22]. As the name suggests, there are two steps:

1. Write rules that generate several models, all of which can be regarded at this point as candidate solutions (described by the corresponding candidate answer sets); this is the "generate" step. For the architectures we've presented, this is the step where we draw **all** possible edges between component ports, without taking into account any constraints.

2. Write all constraints that restrict which of the generated models are valid; this is the "test" step. Clearly this is where we would actually check that no electrical port is connected to a rotational port and other similar constraints.

Additionally, for the more complex problems there can be another step ([21]) called the "define" step, in which we define any intermediate predicates we might need in the other two steps described above. Clearly, these are simply helper rules and **not** constraints.

It now becomes apparent how this method is useful to generate the automobile architectures using answer sets, so let's have a look at the **generator** we've written:

1. We first **define** our components and their ports, along with any auxiliary predicates we may need in the following two steps; for example, this is one way to define a gearset component:

```
type(gs_0, gearset).
rotational(pr_13_0). rotational(pr_14_0). rotational(pr_15_0).
in(pr_13_0, gs_0).
in(pr_14_0, gs_0).
in(pr_15_0, gs_0).
group(pr_13_0, 8).
group(pr_14_0, 8).
group(pr_15_0, 8).
```

   We define its type, its ports (and their types), as well as its single port group; in this case, all 3 ports are part of the same group, therefore can be chosen "freely" for connections.

2. Using a choice rule, we allow for an edge to exist or not between any two ports of used components, thus **generating** a multitude of architectures that need to be filtered based on certain restrictions in the final step. Moreover, the number of electric motors or gearsets that are used is also not fixed, so we must have choice rules for them as well:

```
1 { using_comp(X) : type(X, electric_motor) } 2.
0 { using_comp(X) : type(X, gearset) } 2.
0 { edge(P1, P2) } 1 :- using_comp(C1), using_comp(C2), C1 != C2, in(P1, C
    1), in(P2, C2).
```

3. We define the required restrictions/constraints and **test** which architectures satisfy them. Only those will be considered solutions to our modeling problem and they represent the answer sets of the program. Among the many constraints, we present some below:

```
:- rotational(X), electrical(Y), edge(X, Y).
:- electrical(X), rotational(Y), edge(X, Y).
:- edge(P, P1), edge(P, P2), P1 != P2.
:- edge(P1, P), edge(P2, P), P1 != P2.
:- edge(P1, P), edge(P, P2), P1 != P2.
```

As expected, once all sensible constraints are included, we (eventually) get to the right number of 18 architectures. Interestingly, we don't get the *exact* same ones as the ones from Siemens, but isomorphic equivalents instead. That could mean that we have an equivalent set of constraints, but not exactly the same one (e.g. to avoid generating isomorphic models, we order components of the same type lexicographically; they might order them based on some other rules).

Obviously, any other kind of dataset we may need will be structurally similar to this one, and thus an ASP generator can be easily written in this way, following the generate-and-test technique.

3.3 ILASP

The main learning tool that we decided to use for the objectives of the project was ILASP. Though still under development, it takes advantage of the simplicity and efficacy of ASP and thus seemed like a great candidate to solve the classification problem presented by Siemens.

Its name stands for **I**nductive **L**earning of **A**nswer **S**et **P**rogramming and, thus, it takes advantage of the logic-based branch of Machine Learning using notions of ASP presented above. In order to describe some of its syntax (which, because it uses *clingo* internally, is much the same), its capabilities, limitations and how it generally works, we will mostly rely on its manual, which can be found at [27], along with some observations from [34].

Because ILASP is designed to *learn* certain aspects, it needs two important things:

1. **Background knowledge** (B): its meaning is quite straightforward and refers to any definitions, facts, constraints that the user can provide within the program. This is where the predicates we defined in the generative process become useful.

2. **Examples**: these can be *positive* or *negative*, and are basically **partial** answer sets (thus we provide the program with some examples of what we would and wouldn't want to see in the general solutions/answer sets of the program)

The goal of ILASP is to then find a **hypothesis** H, which is a set of rules, such that $B \cup H$ entails all the positive given examples and none of the negative ones. Of course, its capabilities go far beyond just this, but it's important to understand its basic functionality and purpose. Note that there are also multiple versions of ILASP, designed for different kinds of problems (e.g. some scale better with the number of examples, some deal with inaccurate data more efficiently).

We can see how all this is relevant for the project objective, since we want to learn classifying rules for many architectures based on a much smaller number that have been classified (and can be seen as positive or negative examples for finding a hypothesis describing a certain class, i.e. architectures belonging to that class are the **positive** examples, all other architectures are the **negative** ones).

But before that, we need to formally present some definitions and other aspects of ILASP, especially the ones most relevant to this project:

3.3.1 Learning from Answer Sets

Definitions and examples in this section are based on [34] and [26]. It's interesting to notice that ILASP does pretty much the opposite of *clingo*, since instead of generating answer sets/solutions starting from a problem, it now has a set of partial interpretations of answer sets and tries to find the "problem" that might have these as solutions.

Formally, a **partial interpretation** e is defined as a pair of sets of atoms $< e^{inc},\ e^{exc} >$ and we say that a Herbrand interpretation I **extends** e if $e^{inc} \subseteq I$ and $e^{exc} \cap I = \emptyset$. For example, $\{p,\ q\}$ and $\{p,\ q,\ s\}$ both extend $< \{p,\ q\},\ \{r\} >$, but $\{p\}$ and $\{p,\ q,\ r\}$ don't. Note that a partial interpretation e is **bravely entailed** by a program P if at least one of its answer sets extends e; it is **cautiously entailed** if all its answer sets extend it.

Now we can define more formally what kind of hypotheses ILASP looks for. So given a set of rules B describing the background knowledge, a set of positive examples E^{+}, a set of negative examples E^{-}, a hypothesis space S_M (which is a set of normal rules, choice rules and constraints; the way S_M is constructed is described in the next section), we have to find a hypothesis $H \subseteq S_M$ such that:

- every positive example in E^{+} is extended by at least one answer set of $B \cup H$
- no negative example in E^{-} is extended by any answer set of $B \cup H$

3.3.2 Language Bias

For this section we refer back to the ILASP manual [27] for definitions. As said above, ILASP requires a search space S_M where to look for possible hypotheses. This search space can be specified in its entirety (every possible rule that could be in a solution-hypothesis), but this is obviously not very useful or easy for a user to write. The conventional way to do this is, instead, to specify the **language bias**, which consists of **mode declarations**. There are 4 types of such mode declarations:

- **modeh**: *normal head* declarations;
- **modea**: *aggregate head* declarations;
- **modeb**: *body* declarations;
- **modeo**: *optimization body* declarations.

Each mode declaration can also contain an integer called a **recall** that gives an upper limit for the number of times that mode declaration can appear in each rule. So, for example, a body declaration that says predicate $edge$ of arity 2 can appear in each rule at most twice with two variables of type $port$ is:

$$\#modeb(2,\ edge(var(port),\ var(port)).$$

Before we can consider a full example of language bias, we also need to define what the following two expressions mean:

- $\#maxv$: this restricts the total number of different variables each rule can have;
- $\#max\_penalty$: this restricts the total size of the hypothesis returned.

Other ILASP syntax elements are either intuitive (through their naming) or not relevant. Now consider the following language bias (also from [27]):

```
#modeha(r(var(t1), const(t2)))
#modeh(p) #modeb(1, p)
#modeb(2, q(var(t1)))
#constant(t2, c1)   % constant c1 is of type t2
#constant(t2, c2)   % constant c2 is of type t2
#maxv(2).
```

This is equivalent to the following search space (which would have to be written in full - and this is only a snippet of it - if not for mode declarations, so it's clear why it's always advisable to use language bias):

```
:- q(V1).
:- p.
:- not p.
:- q(V1), p.
:- q(V1), not p.

p.
p :- q(V1).

0 { r(V1, c1) } 1 :- q(V1).
0 { r(V1, c1) } 1 :- q(V1), p.
0 { r(V1, c1) } 1 :- q(V1), not p.
...
0 { r(V1, c2); r(V2, c2) } 1 :- q(V1), q(V2).
0 { r(V1, c2); r(V2, c2) } 1 :- q(V1), q(V2), p.
...
1 { r(V1, c2); r(V2, c2) } 2 :- q(V1), q(V2), not p.
```

3.3.3 Example

At this point we can actually see ILASP in action with an example. We've talked about how the type of hybridization classification is a bit tricky, so we'll leave that for the later chapters, but let's have a look at the easier task: classifying based on axle drive. In other words, let's find hypotheses for **front**, **rear** and **both**.

In order to do that, we need the three main components of the programs: the background knowledge, the set of relevant examples and the language bias. Let's construct the program for obtaining the **front** hypothesis:

1. The background knowledge can contain all defined predicates and constraints from the generative *clingo* program. In particular, because we "suspect", as the user, that the current classification depends on some ports that are used or not, we will need to pay closer attention to the predicate:

```
using_port(P) :- edge(P, _).
```

2. In terms of examples, it's safe to have three examples, one architecture for each class. For the **front** program, clearly only the front-axle architecture will be a positive example, the other two being negatives. Let's have a look at the structure of one example:

```
#pos(c, {front}, {}, {using_comp(ice_0). using_comp(battery_0). using_comp
    (vehicle_0). using_comp(em_0). using_comp(gs_0). using_comp(em_1).
    using_comp(gs_1). edge(pr_15_1,pr_2_0). edge(pe_9_0,pe_5_0). edge(pe_9
    _1,pe_6_0). edge(pr_13_1,fa_0). edge(pr_13_0,pr_10_0). edge(pr_14_0,
    pr_10_1). edge(pr_14_1,pr_15_0).}).
```

   This is a positive example with id c. The first argument is c^{inc}, the second is c^{exc}. The last argument is something called the example **context**, and is no more than further background knowledge that applies to c and c only when computing the relevant answer sets. Clearly, the only "background knowledge" specific for each example is the representation of that example's architecture, which is what we see in the example above.

   Because we've defined the background knowledge so that there are no negated predicates in its rules (i.e. they are **definite**) and because example contexts only contain facts, they form a trivially stratified program and therefore have a single answer set. We should aim to define the mode bias so that $B \cup H$ **remains stratified**. As a result, "at least one answer set" and "all answer sets" would become **equivalent** for all valid architectures, since they would generate exactly one answer set. Hence, in these circumstances, bravely entailing an example is exactly the same as cautiously entailing it.

   So if we look at the *positive* example from above, by definition, *at least one* answer set of $B \cup H$ must entail it. Hence, **if** the hypothesis we find keeps the overall program stratified, then this is the same as saying that *all* answer sets of $B \cup H$ entail it. In particular, *all* the answer sets containing the model representation in the example context **must** entail *front* (i.e. be considered front-axle drive), as opposed to *at least one* answer set, which is the desired stable behaviour.

   For the same reason, we can define negative examples e as positive too, and simply add the *front* atom to e^{exc} instead of e^{inc}:

```
#pos(e, {}, {front}, {MODEL_CONTEXT}).
```

   We do this because ILASP seems to deal with positive examples a little more quickly than with negative ones, so it's nothing more than an optimization from the get-go, using the knowledge of the exact number of answer sets we deal with, namely one.

3. Finally, we need to define the language bias. Clearly, we would like to learn a hypothesis similar to:

   front :- body_of_rule.

   So we expect a head containing the atom *front*. Also, as said before, the body should depend in some way on the predicate $using\_port$, so let's allow it to appear at most 3 times in the body. The language bias should then look more or less like this:

```
#modeh(front).
#modeb(3, using_port(const(portC))).

#constant(portC, pr_2_0).
...
#constant(portC, fa_0).
#constant(portC, ra_0).
...
```

We could define as $portC$ constants only the two vehicle ports, fa_0 and ra_0, but let's not forget that this is a very basic example and in general we would likely not know which ports are relevant and which aren't (a problem we specifically run into in the next chapters, in fact). So for now, let's safely allow all ports to be relevant in the search space and see if ILASP can figure out which ones are the "important" ones.

Also, note how the overall program indeed remains stratified, no matter what hypothesis ILASP learns with this language bias: whether $using\_port$ occurs positively or negatively in a hypothesis rule, its definitions are entirely in the background B, so $B \cup H$ is indeed a valid stratification for the program.

In general, however, we will allow for multiple predicates to be **invented** by ILASP, by which we mean that their definitions are not in the background knowledge, but instead fully contained in the hypothesis. In some cases, that may still allow for an overall stratified program, but in general that does not hold, as is the case with the definitions of the two *invented* predicates, $invPred1$ and $invPred2$, below:

```
invPred1 :- ..., not invPred2, ...
invPred2 :- ..., not invPred1, ...
```

Clearly, there is no way to include these two rules in a *stratified* partition, so in this case brave and cautious entailment would no longer be equivalent. However, if we allow for at most one invented predicate in the mode bias and make it **non-recursive** (i.e. no definition rule of this predicate may contain it in its body), then the set of rules in H can be partitioned into $H = I \cup H'$, where I is the set of definition rules for the invented predicate and H' are the remaining rules in H. Then $B \cup I \cup H'$ is clearly a valid stratification of the program, since rule bodies in I can only contain predicates defined in B; hence, allowing at most one invented predicate means every valid model produces exactly one answer set.

In any case, whenever we allow for at least two invented predicates, $B \cup H$ can become unstratified and we must make all examples negative in order to enforce the "for all" behaviour. For example, a positive example will look like this:

```
#neg(e, {}, {front}, {MODEL_CONTEXT}).
```

This says that no answer set of $B \cup H$ should *not* contain *front*, which is equivalent to saying that all these solutions should contain *front*, as before. A negative example is naturally defined as:

```
#neg(e, {front}, {}, {MODEL_CONTEXT}).
```

This simply says that no answer set of $B \cup H$ should contain *front*, as expected. Even though this is the correct behaviour no matter what the language bias (or background) is, using actual negative examples slows down computation, so should be avoided whenever possible. Even though in general we may allow for more than one invented predicate (thus potentially unstratified programs), we shall only use positive examples in our code listings for ease of notation and interpretation from this point on.

In any case, we now have all three components of our **front** program. We run it (with version 2i of ILASP), and get the output:

```
front :- not using_port(ra_0).

Pre-processing : 0.004s
Solve time : 0.479s
Total : 0.483s
```

Similarly, it learns the following hypotheses for the other two classes:

rear :- not using_port(fa_0).
both :- using_port(fa_0), using_port(ra_0).

These are indeed correct. For another very nice and much more complex example, [27] is highly recommended, where it is shown how ILASP can learn the rules of Sudoku, given only four (partial) snippets of correct and incorrect configurations.

These examples tell us two things in relation to the main goal of this project:

1. ILASP learns underlying rules of an algorithm with relatively little/broad/partial information about the outputs that algorithm can create; this means that it has great potential to create sensible, correct rules of classification for hundreds of thousands of automobile architectures with a very small number of already correctly classified architectures, which is exactly what we want;

2. Even though we don't have to write the full hypothesis space for ILASP, we still have to give quite specific information in terms of what predicates and types of variables and constants to expect in a hypothesis when writing the mode declarations; so a subgoal of the classification task is to not only do the classification correctly, but also as generically as possible, so that it can be reused for other similar (but not identical) kinds of classification problems.

   This is one place where we have to be careful about one of the main goals we discussed at the beginning: never make our solutions specific to one instance of the general problem we're addressing.

3.3.4 Weak constraints

Another goal of this project was to be able to order similar architectures based on a user's preferences. What this means is that, once classification is done, we want to also order architectures within those classes from the most "desirable" to the least "desirable", based again on a small sample of orderings of architectures supplied by the user (in order to get an idea of their preferences). This is where the notion of **weak constraints** comes in.

We again refer to [27] and [25] for definitions (and as always, for more examples and complete explanations these should be consulted).

An **ordering example** is a tuple $o =< e_1,\ e_2 >$, where e_1 and e_2 are positive examples. An ASP program P **bravely respects** o iff it has answer sets A_1 and A_2 such that A_1 extends e_1 and A_2 extends e_2, and A_1 is "more optimal" than A_2. P **cautiously respects** o iff all answer sets that extend e_1 are more optimal than all answer sets that extend e_2.

We write:

$$\#brave\_ordering(order\_name, id_1, id_2)$$
$$\#cautious\_ordering(order\_name, id_1, id_2),$$

where $order\_name$ is the ordering's identifier.

In terms of language bias, the only relevant notions that weak constraints make use of additionally are:

- $\#modeo$, which behaves like $\#modeb$, but refers to the body of weak constraints instead;
- $\#maxp$, which sets an upper bound for how many priority levels a hypothesis can have.

It should be noted that it wasn't exactly clear from the beginning how we might use weak constraints in our implementation and it also wasn't something the Siemens team regarded as necessary for their problem. However, they are a powerful feature of ILASP, and as we've discovered as the project evolved, there are instances where some models require certain kinds of "prioritization", so that's where this notion could come in handy.

3.3.5 Noisy data

Clearly, examples are, in our case, manual classifications. Therefore, human mistakes can (and so will) occur, say a *parallel* architecture is accidentally labeled as *series*. This can lead to one of two scenarious:

1. A wrong hypothesis is obtained. This happens when we do obtain a hypothesis that happens to cover all examples that were provided, but it is a "wrong" hypothesis in the sense that it "sees" a *series* architecture as a *parallel*, so when classifying a large dataset, we can expect a significant portion to be mislabeled.

2. No hypotheses are found. In this case, ILASP simply returns **UNSATISFIABLE**. Unlike the previous case, we are arguably lucky enough to have contradicting examples so that no hypothesis can "explain" both. In this case, we become aware that we either made a mistake or the language bias is not good enough (in that no combination of rules in the hypothesis space can explain all examples).

In case the language bias is not "general" enough, that can be dealt with. For the other case, ILASP can in fact account for user mistakes and still find **the best** hypothesis that explains the given examples, without necessarily covering *all* of them. Wrong examples (i.e. containing mistakes) are called "noisy data", or simply **noise**.

The way ILASP deals with noise is fairly simple (as explained in [27]): instead of trying to cover all examples and find the smallest hypotheses (i.e. the one with the fewest literals) that does so, it expects the user to assign a given weight, or **penalty**, to examples (if an example has no penalty assigned, it is considered noise-free and any computed hypothesis *must* cover it); whenever a noisy example is not covered, we "pay" the penalty for it. For example, this is what the example c we've seen above would look like with a penalty of 10:

```
#pos(c@10, {front}, {}, {using_comp(ice_0). ...}).
```

Now, ILASP looks for the hypothesis which minimizes the sum of the total penalty and the size of that hypothesis. Note how in a completely noise-free program this still holds, as ILASP simply looks for the hypothesis with the smallest size. This is a simple, but effective solution to the problem of human mistake. Given that we will require users to classify fairly complex architectures, this problem is very relevant, so noise is definitely something we must take into account in our implementation.

3.4 Clustering

As already mentioned, if our tool is to select models for the user to classify, a certain strategy is needed. Here we present the motivation why clustering can be used as such a strategy, as well as an "out of the box" algorithm we make use of later on, before implementing our own techniques of clustering.

3.4.1 Motivation

Even with the initial datasets we received, one other problem became very clear: some labels may correspond to a majority of the given architectures, while some labels may only correspond to very few models. That means that completely random model generation for manual classification might not work very well. Since in this case every architecture has an equal probability of being chosen, if there are, say, 3 possible labels and one label corresponds to only 1% of the models, then there is a 1% chance of getting one such model. That probability will roughly stay the same throughout time, since the overall distribution of the three labels will remain more or less the same on the remaining set of unlabeled architectures.

To give a quick proof for this, let there be N initial models, 1% of which have label a, and after some time M models have been (completely randomly) selected and labeled. As a result, we can expect roughly the same distribution over those M architectures too, as this is equivalent to random sampling M models from a list of N. This means around 1% of the M models have label a, so simple arithmetic yields that the remaining $N - M$ models still contain around 1% models with label a (this is especially true for large N and M). So basically at each point in time there is an almost constantly small chance of getting those labels that are assigned to very few models. Since we would like to learn hypotheses for **all** labels, we need some sort of strategy to try to remove the bias of the model-to-label distribution.

That's why we introduced a heuristic that aimed at clustering "similar" models together. The rationale is that this clustering should group together as many models with the same label as possible, so that if one label is seen in a vast majority of architectures, we no longer randomly choose **models**, but **clusters** instead, so each of these "biased" models gets a smaller probability of getting chosen, while the models in smaller clusters get a larger probability.

Specifically, consider the following example: we have 500 models, 1 of which has label a and the rest have label b. Normally, there's a chance of about 0.2% that we get the one with label a if we select models completely at random. However, if we create two clusters, one with the a-model, along with 9 misplaced b-models, the second with all the other models, then the chance of getting the architecture with label a is now the probability of selecting the first cluster times the probability of selecting the right model from that cluster:

$$\begin{aligned} P(A\ selected) &= P(A\ selected \cap C_1\ selected) \\ &= P(C_1\ selected) \cdot P(A\ selected \mid C_1\ selected) \\ &= \frac{1}{2} \cdot \frac{1}{10} \\ &= 0.05 \\ &= 5\% >> 0.2\%, \end{aligned} \tag{3.2}$$

where A is the a-model and C_1 is A's cluster. Note that A is selected iff A and C_1 are selected (since A is in C_1 only), which explains the first equality from above. We see a large boost in A's chances of getting chosen now, even with some misplaced models, while still having the rest up to 100% (i.e. 95.5%) chances of getting a b-model still. While, in general, there would be more than just one model for a label, and thus better chances than $< 1\%$, this was only to prove the mathematical reasoning behind why clustering can be useful.

The only issue here is that we cannot cluster models based on their corresponding labels; those are unlabeled architectures, and learning their labels is what the whole purpose of this project is in the first place. So we can only try to cluster them based on some features that *might* be relevant, so that hopefully a lot of the biased models will be put in the same clusters.

Based on the data that we've worked with so far, there seems to be some empirical evidence that labels are somewhat connected to the types and number of components that appear in models. What this means is that some components are likely to be more relevant to some labels and potentially less relevant to others. Also, some labels seem to be more present in models with fewer components, while others are seen more often in the more complex architectures. Just to be clear, this, in general, is not true for classifying typed graphs, as the labels corresponding to them can be virtually anything (they can even be pre-assigned at random), this is only an observation we made based on the data from Siemens.

3.4.2 MeanShift algorithm

Regardless, with this heuristic in mind, there are two main candidate algorithms with the behaviour we've described so far: **MeanShift** and **K-Means**. Despite the appeal and popularity of the latter, its main problem is its requirement for the number of clusters to look for. In our case, "guessing" or "approximating" the number of possible clusters would be another big variable in a problem already based on heuristics (so on more guessing). That is why we decided to go for MeanShift instead, since it doesn't have that requirement.

This algorithm looks for "blobs" of **close points** in a dataset (where points are **N-dimensional vectors**), computes for each such blob a **centroid**, which is the *mean* of the points in that blob (which explains the name of the algorithm, since these means are shifted until they stabilize); the resulted clusters are basically the blobs of points. More details about how the algorithm works and some other applications can be found at [17], [7] and [5]. For the purposes used in our project, this much information about the algorithm will suffice.

One thing that could be worrying is exactly MeanShift's ability to compute the clusters without requiring their number: what if we generate hundreds of thousands of clusters? Then generating a random cluster is almost the same as generating a random model, and we're back to square one (note that this wouldn't affect the overall program's correctness; it would only affect one of the strategies to optimize computation). The thing is, as explained before, the very large number of architectures is usually not due to a large number of *components*, but instead because of all the ways in which *edges* can be generated between the *ports* in that architecture. Because we don't cluster based on ports or edges, having a relatively small number of components (must be less than 64, will see why later on, and is usually much less than that anyway), there won't be that many expected clusters. For example, one dataset from Siemens contained more than 170,000 models, but was generated with only 10 distinct types of components. This means that models vary from one another mostly on "how" their components are connected, and not so much in terms of their actual compositions, so the number of clusters will likely remain relatively small.

In order to use MeanShift, which requires N-dimensional vectors, for some N, we choose N to be the total number of types of components in our problem, and the vectors can be frequency vectors for the components for each model, i.e. the i^{th} component of a vector of a model is the number of components of the i^{th} type that model contains. For example, if all component types in our problem are (in this order):

$$(\text{vehicle, battery, ICE, electric\_motor, gearset}),$$

then for a model with **one** vehicle component, **one** battery, **one** ICE, **two** electric motors and **no** gearsets, the corresponding vector is:

$$(1,\ 1,\ 1,\ 2,\ 0).$$

The initial dataset from Siemens with the 18 hybridization models yields the following **centroids** (as N-dimensional points), when ran with MeanShift:

```
[[1. 1. 1. 2. 1.]
 [1. 1. 1. 2. 2.]
 [1. 1. 1. 1. 2.]
 [1. 1. 1. 2. 0.]
 [1. 1. 1. 1. 1.]
 [1. 1. 1. 1. 0.]]
```

Listing 3.1: Hybridization MeanShift centroids

Note how these are exact, i.e. the centroids coincide with actual provided composition vectors. This is not going to be the case in general, but it happened in this case because of the very small number of models in the input. As a result, each cluster will contain architectures with *exactly* the same components (but obviously connected differently). However, running the algorithm with a different dataset, we find one centroid equal to:

```
[1. 1. 1. 0.16666667 0.16666667 0.5 0. 0. ... 0.16666667 ...]
```

Listing 3.2: A non-exact MeanShift centroid

Clearly, this is an example of a non-exact centroid, meaning that architectures with different (but very similar) compositions are clustered together, which is the desired behaviour.

Finally, the important output from MeanShift are actually not the centroids, but the labels corresponding to each model. For the 18 hybridization models, we get the following output:

```
[3 1 3 4 2 1 4 1 0 1 2 0 2 0 5 0 5 0]
```

Listing 3.3: Hybridization MeanShift labels

Each of the elements in this list is the *index of a cluster*, and thus all elements with the *same value* are in the *same cluster*. Note that this output is in the same order as the models that were provided as input to MeanShift. So according to this output, the first model belongs in the cluster with index 3, the second - in cluster number 1, and so on. Using this, it's now easy to construct a map from cluster ids to lists of models in the corresponding clusters. Moreover, it's worth noting that the cluster indexes also correspond to the centroids listed above and as a result we know that, for example, the first model in the list is "very close in composition" (identical, in this particular case) to the corresponding centroid of index 3 with composition:

```
[1. 1. 1. 2. 0.]
```

Listing 3.4: Hybridization MeanShift centroids

Although the user will never be aware of these clusters specifically, it's worth knowing how the models are "grouped" together when going in the further steps of development.

3.5 Related existing works

This project *builds* onto an already heavily researched field called **Computational Design Synthesis** (explained in more detail in [16]). It refers to the automation of generating all models that abide by a given set of constraints/restrictions, which obviously has applications in various different domains, from Chemistry and Biology (e.g. [23], [32]) to Robotics (e.g. [29]), Mechanical Engineering (as is the case proposed by Siemens) and many others. One of the biggest problems of Computational Design Synthesis is that it very often is the case that the given constraints are not completely exhaustive in order to remove all unwanted models (some constraints might be more subtle and only intuitively applied by the engineers when evaluating a model manually, so they are not specifically used in the generative process). As a result, the generated models are not exactly *solutions*, but rather *candidate solutions* that often have to be further inspected by an expert in order to be accepted or dismissed. This brings us back to the main problem this project is trying to solve, which is classifying those generated candidates (first into valid/invalid and then into more specialized classes).

In general, there are some textbook Machine Learning methods to do certain kinds of classification, depending on the specific problem they need to solve. One way to do this would be to somehow use neural networks on the graph representations and hope to learn how to classify them based on a subset of already classified graphs.

The trouble with neural networks is that generally quite a large amount of classified training data is needed (often of the order of hundreds of thousands) before the network can reliably be used for classification, which in our case is a big problem, as we don't have that much data to work with; given the complexity of the graphs the user needs to classify, we cannot and should not expect a massive amount of training data from them.

That is why we instead researched how to do classification using **Inductive Logic Programming** (ILP). Even within this field there are several different approaches in order to obtain a valid hypothesis for a set of positive and negative examples (along with some background knowledge), and obviously all such systems come with goods and bads, and thus some are more appropriate for our project than others.

One method used by a lot of modern ILP systems is **inverse entailment** ([34], [4]), which refers to the following property for any background knowledge B, hypothesis H and example E:

$$B \wedge H \models E \Leftrightarrow B \wedge \neg E \models \neg H,$$

so in order to obtain a valid hypothesis, we add to the background knowledge the negation of an example and generate (the negation of) a valid hypothesis, which would entail that example. Even this process of finding the hypothesis based on one example can be divided in two big categories ([14], [37], [20]):

- **bottom-up** approach, where the hypothesis starts with a "most specific" rule that explains the example and usually generalizes the rule as much as possible (too general a rule might also entail some negative examples); some representative such learners are Golem and more recently ProGolem [30];

- **top-down** approach, which is obviously opposed to the previous one in that it starts with a very general rule and specializes it until no negative examples are entailed (and all positive ones are, still); the most popular such learners at the moment are Aleph [36] and FOIL (along with several enhanced versions of FOIL) [24]

ILASP takes a different approach than these by using Answer Sets in order to generate a valid hypothesis. This allows it to, for example, not only handle noisy data quite well, but also to generate weak constraints based on orderings of partial interpretations, so that it is an ideal candidate to solve the classification problem presented by Siemens.

Once classification is done with some of the provided classified architectures using ILASP, we need to develop the tool that eases the classification process for the engineers. A formal discussion on how this might be done is presented in [15], where they list two different approaches:

- **passive learning**, where the user (in our case the engineer/designer) supplies the tool with a number of manually constructed and labeled architectures, which the tool then uses as positive (or negative) examples in order to learn the classification hypotheses. This obviously works initially for our tool, in order to test that the classification algorithm does work with the already labeled data that we have.

  However, when we have hundreds of thousands of architectures to classify, it becomes virtually impossible for a user to supply a reasonable number of labeled models (that cover as many different classes as possible), some of which might be very complex graphs, so this method becomes infeasible for our tool in the long run;

- **active learning**, where the learning process is much more interactive. Instead of requiring the user to come up with the pre-labeled examples, the tool generates (partial) interpretations itself and gives them to the user, who only has to do the labeling; this back and forth process goes on until classification is complete.

  One big advantage of this approach is that the tool can choose what examples to give to the user based on a certain strategy, rather than randomly, which speeds up the learning process and minimizes the number of examples the user has to classify. One such tool is

QuAcq [15], but according to [18], it takes it more than 8000 classified partial interpretations of a 9X9 Sudoku board before it can learn its rules (constraints).

On the other hand, ILASP only needs 4 partial interpretations (1 positive and 3 negative) in order to learn the rules for an (admittedly smaller) 4X4 Sudoku board [27], which again suggests that Learning from Answer Sets has great potential with little information when done more or less optimally (i.e. when choosing examples strategically, so that the user doesn't label redundant interpretations).

Finally, another interesting and relevant paper is also [33], which shows how ILASP's ability to learn weak constraints can help machines reason about something as complex as human behaviours in a more natural, human-like way. This is important for our project because we wouldn't want a random (possibly very specific/over-fitted) sequence of rules to describe when an architectures is "better" than another, but rather have rules ordered by priorities and descriptive enough to be human-readable.

Chapter 4

Passive learning of class hypotheses

4.1 Initial attempts

As said in the introduction, one of the main objectives of this project was to develop a tool to make it **easy** for an engineer to classify architectures. This must, consequently, satisfy a few requirements:

- it must be **intuitive**: this means that an engineer who wants to classify a set of architectures should not need any prior knowledge of ASP/ILASP/*clingo*; all of this should happen "under the hood", while the user should only have to give the actual labels for a subset of architectures that the tool chooses.

- it must be **efficient**: the tool itself should not create too much overhead and the main part of the computation time should correspond to ILASP's actual learning process and perhaps the final classification (given the large dataset it would involve).

Note that, in a sense, the learning tool is actually quite independent of the actual hypothesis learning process. The tool should be thought of as a **cycle of active learning**; it generates architectures (according to a certain strategy), displays them to the user who is prompted to classify them, learns hypotheses that explain those classifications, and then repeats, until the user is satisfied with the result. We've seen an overview of this in 2.3.

We shall go into more detail of how this cycle works in other chapters, but it's worth noting that exactly **how** those hypotheses are learned is not relevant to the tool, only that they are indeed generated in order to be shown to the user, so that's why the *cycle* and the actual *hypothesis learning process* are two separate tasks in terms of implementation and we can discuss and analyze them separately.

Before creating the tool as an active learning task, we needed to make sure that ILASP can actually generate valid, sensible hypotheses for the hybrid architectures case, so we initially settled for passive learning, with one ILASP program that was already provided with sufficiently many labeled architectures and see what hypotheses it comes up with. This was useful for several reasons:

- easier to test: any error/odd result is definitely located in the ILASP program and not in the "wrapping" cycle, so it can be found and corrected much more easily.

- easier to reason about: when we have a fixed set of known examples to be provided to ILASP, it becomes a lot easier to come up with the predicates that would be part of the hypothesis space.

- easier to analyze its efficiency: since we only run one task, directly from the command line (and not from within another script), we can see exactly how long that task takes and how its complexity changes when tweaking some of the variables it depends on.
- sanity check: it provides reassurance that an incremental learning tool could work; in theory, such a tool should eventually be able to converge to a valid hypothesis if one exists in the passive learning case, but if no satisfying hypothesis can be found in this case, than an iterative tool likely has no "good enough" hypothesis to converge to at all.

Note that all these points remain valid even after an active learning tool is created, and whenever we needed to implement a new feature inside the actual ILASP learning process of the tool, we first tried it with a passive-learning ILASP task, for all the reasons above. Only if/after it worked there properly did we implement it in the actual tool.

As said before, the data we were provided with contained a lot of redundant information (redundant for the classification process), so in order to parse it we initially wrote a basic pre-processor using NodeJS. The outputs could then be used as contexts for the examples in the ILASP task.

So let's say we want to find a hypothesis for parallel architectures. Then one way would be to define an atom $parallel$, and try to find a hypothesis that comprises of rules of the type:

parallel :- body_of_rule.

What this says is that $parallel$ holds if $body\_of\_rule$ holds for at least one of the definition rules in the hypothesis. Note that we can have several such rules in one hypothesis, meaning that an architecture can be $parallel$ for several reasons, and we only need at least one such reason to hold.

However, in order to be able to find such rules, we also need to indicate in each provided example whether that example corresponds to a parallel or non-parallel architecture. For example, this is what an example would look like for a parallel architecture:

```
#pos({parallel}, {}, {
comp(n1,ice).
port(n1,pr_2_0).
comp(n2,battery).
port(n2,pe_5_0).
port(n2,pe_6_0).
comp(n3,vehicle).
port(n3,fa_0).
port(n3,ra_0).
comp(n4,em).
port(n4,pe_9_0).
port(n4,pr_10_0).
edge(pr_2_0,ra_0).
edge(pe_5_0,pe_9_0).
edge(pr_10_0,fa_0).
}).
```

The notions used here are quite intuitive. The example context contains, as usual, the components (with their ports) of the corresponding architecture, along with the edges that connect them together.

For each component predicate we provide the component id and the component type (e.g. $comp(n1, ice)$ means component with id $n1$ has type ice). For each port predicate we provide the (parent) component id and the port id (e.g. $port(n1, pr\_2\_0)$ means port with id pr_2_0 belongs to component with id $n1$). For each edge predicate we provide the ids of the two ports connected by the edge (e.g. $edge(pr\_2\_0, ra\_0)$ means that the port with id pr_2_0 is connected to the port with id ra_0). These will remain the standard notations throughout the report from now on, unless otherwise stated.

One thing to note here is that some information that is structurally relevant in the architecture **generation** process (e.g. port group, direction) has been parsed out. The reason for that is that they are simply not relevant to the **learning** process; they were vital for removing invalid or (isomorphic) duplicate architectures from the generated architectures, but they don't provide any useful information in terms of common substructures of different architectures that are not isomorphic, so they won't help with classification.

Lastly, the example is labeled as "parallel" by adding the atom $parallel$ to the first argument of the example, as per the discussion in 3.3.3. Similarly, a negative example (in this case corresponding to a *series* architecture) would be written as follows:

```
#pos({}, {parallel}, {
comp(n1,ice).
port(n1,pr_2_0).
comp(n2,battery).
port(n2,pe_5_0).
port(n2,pe_6_0).
...
edge(pr_2_0,pr_10_0).
edge(pe_5_0,pe_9_0).
..
}).
```

We don't *care* that this is explicitly a "series" architecture, only that it is not a "parallel" one, since we are only computing a hypothesis for parallel architectures in this ILASP task. Hence, we are adding the atom $parallel$ to the second argument of the example.

Given that most of the information corresponding to each architecture is fully captured by the example contexts, the background knowledge is quite small and no longer contains definitions of components (e.g. gearsets) as before:

```
compV(C) :- comp(C, _).
edge(P1, P2) :- edge(P2, P1).
are_connected(C1, C2) :- port(C1, P1), port(C2, P2), edge(P1, P2).
```

It merely creates a unary predicate $compV$ to define the component variables using the binary $comp$ predicate from the contexts (e.g. $compV(n1)$, so this tells us *which* atoms are the components, dropping their type). It also makes sure all answer sets "see" the graph edges as undirected (or rather bidirectional), and finally it defines component-level connections through the predicate $are\_connected$: basically, an edge between the ports of two components means those components *are connected*.

Having this, we can now feed as many of the architectures from the dataset in the ILASP passive learning task (as positive or negative examples, depending of what class hypothesis we want to learn). Now, the hard part is to actually define the hypothesis space. This is tricky for several reasons:

- it needs to contain rules that are **"specific"** enough to be able to rule out all the negative examples
- it needs to contain rules that are **"general"** enough to entail all the positive examples as well
- it must not be "too large", since the ILASP computation time depends heavily on that

- it must be defined in terms of predicates that can be used for the general case, not for specific cases (e.g. hybridization type)

For these reasons, and especially the last one, we tried to define the hypothesis space first without looking at the "hybrids" case too heavily; in other words, we didn't want to manually compute the hypotheses ourselves for all the labels and then be biased by those when building the hypothesis space. That is why we initially defined the mode bias more or less in the following way:

```
#modeh(parallel).

#modeb(3, are_connected(comp(compV), var(compV))).
#modeb(3, comp(var(compV), const(compC))).

#constant(compC, ice).
#constant(compC, vehicle).
#constant(compC, battery).
#constant(compC, em).
#constant(compC, gs).

#modeh(p, (non_recursive)).
#modeb(p).
```

Basically, all rules will ideally have $parallel$ as their head and their bodies in terms of component connections. We also use p for **predicate invention**, i.e. its definition is not in the background knowledge and can be entirely set by ILASP in the hypothesis. Note that this hypothesis space contains "high-level" rules only, in that it does not consider ports and edges, only component connections. This seems reasonable and closer to what a human-being would intuitively consider when looking at a diagram of an architecture.

However, no matter how we tweak the variables in this hypothesis space, we cannot get valid hypotheses other; we always get UNSATISFIABLE. Note that positive examples alone could be entailed by the hypothesis:

```
parallel.
```

Negative examples alone are entailed by the empty hypothesis (since they aren't actual negative examples, but still positive examples that simply should not "generate" $parallel$ in their answer sets, which an empty hypothesis does not, as required). However, when we have both positive and negative examples, we immediately get UNSATISFIABLE with this mode bias.

This meant that we had to analyze the hybridization models more closely and try to find proper predicates that can generate valid hypotheses, while still general enough and independent from the hybrids case. Namely, we had to find a way to generalize simple component-connections to **substructures** that recur within the same class.

For example, if we look at why an architecture is parallel or series, it's usually because of how the **battery**, **ICE** and **vehicle** are (eventually) connected together. In a series architecture the ICE does not "directly" power the vehicle axles, and instead the battery (and thus the electric motors) are the ones responsible for that. One thing to mention is that we don't know the exact physical implications and engineering details, we only asses the graphs by pure connectivity observations, as an automated (generic) tool should.

Here is a sample diagram of a series architecture, provided by Siemens. The red component is the ICE, the green component with the slightly longer text is the battery (the other two are the electric motors), the black component is a gearset and the blue component is the vehicle. We can, indeed, observe how the ICE and the battery are in a "series" order before "entering" the vehicle:

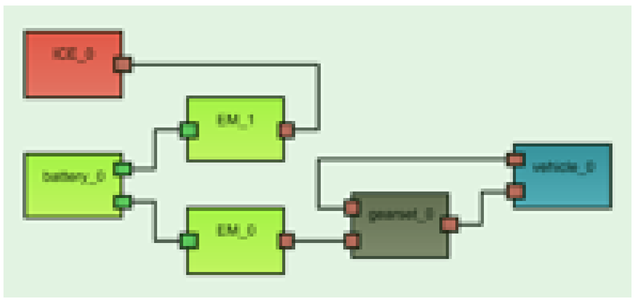


Figure 4.1: Series architecture

In other words, if we disregard the other components, this architecture can be reduced to this "high-level" form:

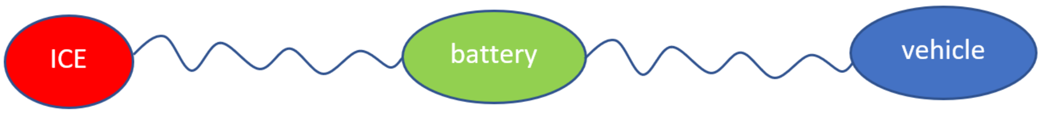


Figure 4.2: Series "high-level" view

Note that the "squiggly" line means that there is a path that starts from one component and "eventually" gets to the other. For simplicity, we can assume that whenever we use this type of line, it does not contain any cycles. In other words, a squiggly line between components A and B represents a **basic** path (i.e. with no cycles) from A to B. So the figure above says that there is such a basic path from the ICE to the vehicle that passes through the battery.

In parallel architectures, on the other hand, both the ICE and the battery directly contribute to powering the vehicle axles. Here are two samples of parallel architectures, once again provided by the Siemens team:

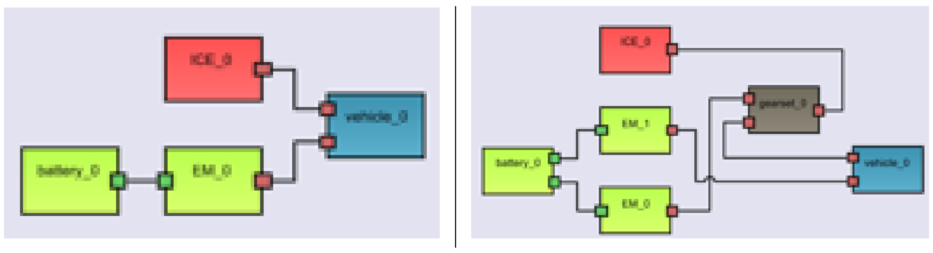


Figure 4.3: Parallel architectures

Now both of these can, as above, be reduced to a high-level form by removing irrelevant components:

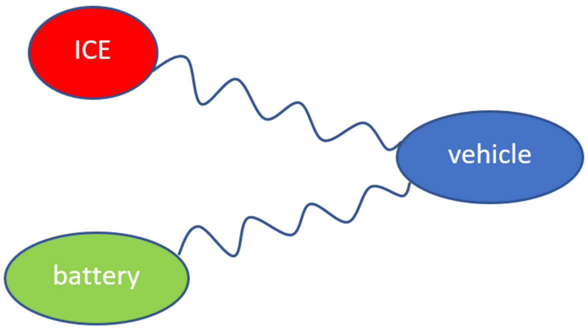


Figure 4.4: Parallel "high-level" view

The series-parallel ones are a bit more complicated to reason about, but before we can deal with them, ILASP should be able to at least differentiate between a parallel and a series architecture, so we need to formally define what the squiggly line means in ASP syntax, as it is definitely relevant to what the hypotheses might contain. The problem with this is that there is no formal way to represent lists or sets, and we do need this in our case. As said before, a basic path must not have any cycles, so we must keep track of the nodes it contains and make sure there are no duplicates.

Note that, although we initially required the squiggly line to not contain any cycles for simplicity only, we actually now require this in order to differentiate series from parallel architectures. Indeed, if that wasn't a requirement, then we can represent a series structure as a parallel one in the following way:

$$\text{ICE} \leftrightsquigarrow \text{battery} \leftrightsquigarrow \text{vehicle} \leftrightsquigarrow \text{battery}$$

So although we have a series architecture (because of the series ICE - battery - vehicle), we go from the ICE to the battery, then to the vehicle, and then "back" to the battery. But if we ignore the first "battery" in the path above, we get:

$$\text{ICE} \leftrightsquigarrow \text{vehicle} \leftrightsquigarrow \text{battery}$$

This is a parallel structure (as per Figure 4.4). We can similarly show that a parallel structure will be seen as a series one also, if we don't consider the squiggly line strictly non-cyclic. So we must define it in such a way that it contains no cycles in order to properly differentiate series from parallel structures.

To summarize, it becomes clear that we need some way to define lists, or at least sets, in our programs, if we want to define basic paths between components.

4.2 Using binary component values to define patterns

One way to do this is to assign to each component a different bit (e.g. ICE - 0001, vehicle - 0010, battery - 0100, etc). This way, a **set** of components can be uniquely and clearly represented by a binary value comprising of all the bits of the elements of that set (e.g. with the bit values from above, the set {ICE, vehicle} is encoded with the value 0011 = 0001 | 0010, the set {ICE, vehicle, battery} is encoded with the value 0111, etc).

This also shows us how to do some of the common set operations:

$$\text{"}\{\text{ICE, vehicle}\} \cup \{\text{ICE, battery}\} = \{\text{ICE, vehicle, battery}\}\text{"} \Leftrightarrow \text{"}0011 \mid 0101 = 0111\text{"}$$
$$\text{"}\{\text{ICE, vehicle}\} \cap \{\text{ICE, battery}\} = \{\text{ICE}\}\text{"} \Leftrightarrow \text{"}0011 \,\&\, 0101 = 0001\text{"}$$

So we can use these to update our ILASP program. First, we need to add the bit-values for each component to every example context, like so:

```
#pos({parallel}, {}, {
comp(n1,ice). val(n1, 1).
port(n1,pr_2_0).
comp(n2,battery). val(n2, 2).
port(n2,pe_5_0).
port(n2,pe_6_0).
comp(n3,vehicle). val(n3, 4).
port(n3,fa_0).
port(n3,ra_0).
comp(n4,em). val(n4, 8).
port(n4,pe_9_0).
port(n4,pr_10_0).
edge(pr_2_0,ra_0).
edge(pe_5_0,pe_9_0).
edge(pr_10_0,fa_0).
}).
```

Note that we don't explicitly use binary representations, but rather assign a power of 2 instead. This means that this approach works only for architectures with at most 64 components (since that is how many bits integers use in ILASP). This seems reasonable, as for architectures larger than that it becomes quite unfeasible for ILASP to learn something in a decent amount of computation time.

Having these values, we can now define basic paths.

```
basic_path(C1, C2, M + N) :- are_connected(C1, C2), val(C1, M), val(C2, N).
basic_path(C1, C2, M + N - P) :- basic_path(C1, C3, M), basic_path(C3, C2, N),
    val(C3, P), M & N == P.
```

The first definition is the base case. Having two connected components, the basic path from the first to the second is encoded to the sum (equivalent to the OR) of the values of the two components. The second definition is the generalization: having a basic path from component C1 to C3 with code M and a basic path from C3 to C2 with code N, if C3 is the only intersection of those paths (i.e. if M & N == P, where P is the value assigned to C3), then there is a basic path from C1 to C2, and it is encoded to M + N - P (equivalent to M OR N, since M and N both "contain" P, so M + N will "contain" it twice).

Note that we used "+" instead of bitwise OR because some such operations were not yet added to the ILASP parser, but could easily be replicated in the ways described above.

One thing to note is that, if $basic\_path(C1, C2, P)$ holds, it means that there exists **at least one** basic path with code P *starting* from $C1$ and *ending* at $C2$. That does not mean, however, that **all** basic paths with code P start from $C1$ and end at $C2$; P only encodes a **set**, not a list. Take, for example, an architecture with three components: a, b and c such that any two components are connected to each other. Assume a is encoded to 001, b to 010 and c to 100. Then all the following basic paths will be encoded to 111 (7 in decimal):

$$a - b - c : basic\_path(a,\ c,\ 7).$$
$$b - c - a : basic\_path(b,\ a,\ 7).$$
$$c - a - b : basic\_path(c,\ b,\ 7).$$

Also, it's worth noting that the predicate $basic\_path$ is clearly symmetrical in its first two parameters, since the graphs are undirected.

Now, using this predicate, we can actually define a (first) proper **pattern**. As pointed out in Figures 4.2 and 4.4, we need to have the notion of a basic path between two components that also passes through a third. This simulates what an engineer would look for in a graph:

among all the connections, they would search for a certain pattern; for now, we restrict that pattern to "basic path from A to C, going through B" and we shall define some other more complex patterns later on.

We can formally define this notion as a **direct_path**:

```
direct_path(C1, C2, C3) :- compV(C1), compV(C2), compV(C3), C1 != C2, C2 != C3,
    C1 != C3, basic_path(C1, C2, M), basic_path(C2, C3, N), val(C2, P), M & N
   = P.
```

There doesn't seem to be a point in defining *direct_path* when two or more of the components coincide, so we can require all three to be mutually distinct. We then require a basic path to exist from C1 and C2 and another from C2 to C3, and finally we require the intersection of those two basic paths to be C2 only. This makes sense and it's perhaps easier to reason about visually, using Figure 4.5:

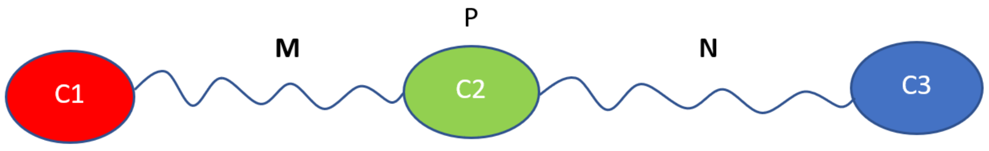


Figure 4.5: direct_path diagram

4.3 First hypotheses

Having this, we can give ILASP another try, this time defining the mode bias in terms of the more general predicate direct_path instead of are_connected:

```
#modeh(parallel).

#modeb(1, direct_path(var(compV), var(compV), var(compV))).
#modeb(3, type(var(compV), const(compC))).

#constant(compC, ice).
#constant(compC, vehicle).
#constant(compC, battery).
#constant(compC, em).
#constant(compC, gs).

#modeh(p, (non_recursive)).
#modeb(p).

#bias(":- body(direct_path(V, _, V)).").
#bias(":- body(direct_path(V, V, _)).").
#bias(":- body(direct_path(_, V, V)).").
```

The last three lines from above are further restrictions on the hypothesis space: there is no need to generate rules in which at least two variables in $direct\_path$ are equal, since by definition this is not possible. This will add to the pre-processing time in ILASP, but will greatly improve the actual computation time.

This time, even with a substantial number of examples (9 out of 18), both positive and negative, we no longer get UNSATISFIABLE. Instead, we get the following hypothesis, after about 10 seconds:

```
% PARALLEL
p :- direct_path(V0, V1, V2), comp(V1, battery).
parallel :- not p.
parallel :- direct_path(V0, V1, V2), comp(V1, vehicle).
```

This is indeed very close to what we predicted, as it provides both the rule to "accept" parallel architectures directly (the one where the vehicle is "in the middle" of a direct path), but also the one that "rules out" series architectures, where we generally have the battery "in the middle" of a direct path (by defining this using the invented predicate p and then using its negation). Similarly, we run ILASP for the series and series-parallel hypotheses (by simply switching which examples to be considered positive and which negative), and obtain:

```
% SERIES
p :- direct_path(V0, V1, V2), comp(V1, vehicle).
series :- direct_path(V0, V1, V2), comp(V1, battery), not p.
p :- direct_path(V0, V1, V2), comp(V0, gs), comp(V2, vehicle).

% SERIES-PARALLEL
series_parallel :- direct_path(V0, V1, V2), comp(V1, battery), p.
p :- direct_path(V0, V1, V2), comp(V0, gs), comp(V2, vehicle).
```

Note that all these hypotheses were obtained in *different* runs of ILASP, and so they contain independent definitions for the invented predicate p, e.g. the definition for p in the parallel hypothesis should be ignored when considering the series hypothesis.

Again, we can see how these hypotheses are somewhat consistent with our initial assumptions, and that we even get a hypothesis for the series-parallel architectures, even though we didn't closely analyze them. That suggests that our definition for the pattern $direct\_path$ is not too specialized, and rather describes something generally recognizable in any undirected graph.

Finally, we can now take these hypotheses and add them to our generation program (with distinct invented predicate names for reasons discussed above), and run the generative program with *clingo*. The output is quite large and we will not display it here, but it's worth mentioning that the hypotheses seem to be mutually exclusive (in that no architecture has more than one label), which is in general not to be expected, since they come from separate ILASP programs. Those programs don't (and can't) require them to be completely mutually exclusive, only to be so on the set of examples they share (i.e. a positive example for one label must be negative for all other labels, and vice versa).

Also, using the diagrams provided by Siemens, we can see that with these hypotheses, all architectures are indeed classified correctly. This is promising for several reasons:

- as said before, if it works for passive learning, it should also work for active learning and obtain valid hypotheses there, but potentially more quickly and in an interactive way.
- although we chose to use 9 examples out of the 18, this was to make sure we don't get UNSATISFIABLE; upon inspection, we ca see that only 3 examples are needed for each task in order to learn completely accurate classification hypotheses, which is indeed very promising for larger datasets.
- we were also able to obtain valid hypotheses with a relatively small hypothesis space. Keeping the hypothesis space as small as possible will be vital for the tool; here we only had 5 types of components, but this number can be much larger than that, so we need to restrict the hypothesis space as much as possible as that number increases, as well as the number of patterns we define, apart from $direct\_path$.

Chapter 5

Basic active learning tool development

Once a passive learning ILASP prototype is functional, it can be integrated into an active learning cycle, and in this chapter we see how the basics of this cycle are implemented. As said before, the cycle and the ILASP implementation must be more or less independent, and so any developments of the cycle discussed now in no way affect the ILASP hypothesis learning capabilities or vice-versa. It's worth remembering the high-level structure presented in 2.3:

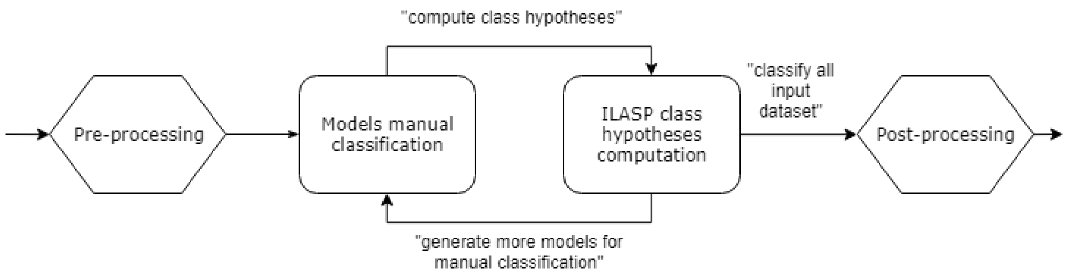

"compute class hypotheses"
Pre-processing
Models manual classification
ILASP class hypotheses computation
"classify all input dataset"
Post-processing
"generate more models for manual classification"

Figure 5.1: High-level learning tool structure

For the initial attempts described before, we used NodeJS for pre-processing. Initially, we tried to use NodeJS for the whole cycle, given the substantial number libraries it has to offer, but eventually we settled for Python; it simply provides more tools in terms of UI, which, as we shall see, is an important part of the tool - *classilasp*.

In terms of implementation, the main goal was for the user to have to know as little about ASP/ILASP as possible, and as a result the tool should do all the pre-processing of the dataset, analyze it, and only give the user diagrams to be labeled. That also raised the question of what format the tool should expect the for the input file. Siemens initially provided us with xml and JSON formats, but then they agreed to generate them in Prolog syntax. Given the similarity of syntax between Prolog and ASP, we decided to let the tool accept this format and use it exclusively for pre-processing, at least until pre-processors for other formats are implemented.

After pre-processing should come the actual "back and forth" cycle of providing the user with some unlabeled models, allow them to inspect them and then store the label they've selected. There are some questions to be raised here as well:

- What's the best way to **separate** the pre-processing work from the active learning cycle?
- What's the best way to **display** the models for the user? How much information should a diagram contain in order to allow the user to label it without becoming overly complicated?

- Should the diagrams show only connections between components if a **high-level** classification is expected and more in-depth ports and edges information if a **low-level** classification is expected instead?
- How reliable should the user labels be considered? In other words, should we allow for mistakes or subjective answers as opposed to full-proof, objective labels? This decides whether the ILASP tasks should consider using **noise** or not, which obviously affects the computational time, as well as the hypotheses they might produce.
- Given that the main part of the tool's computation time is taken up by running the ILASP tasks, **how often** should they be run? Should it be up to the user when they are run or should the tool decide? Can the tasks be run independently, and thus in **parallel**?
- Since the ILASP programs have to now be constructed by the tool, how much information should the tool know about the actual ILASP implementation before they become too inter-dependent? How can the implementations be correctly and clearly **separated**?
- When does the cycle **end**? Is there a point when the tool can decide that classification is completed or is that up to the user?

If these questions are dealt with correctly, a set of hypotheses should be produced and they can be used for the final step: classifying the entire initial data based on these hypotheses. For symmetry reasons, we named this step post-processing, given that we again have to go through all the given architectures, process them and decide what their corresponding label is, based on the computed hypotheses.

It's important to note that the quality and correctness of this classification step depends entirely on how "good" the hypotheses we have are. In other words, if we get mislabeled (or unlabeled, or even multiply-labeled) architectures, that means that the classification cycle was not 100% accurate. This can be explained by several reasons:

- not enough data was labeled before exiting the cycle
- the tool didn't choose different enough architectures to be labeled, and so the hypotheses could not cover all reasons explaining a certain label
- the user made one or more mistakes and noise was not used or it was, but couldn't account for the amount of mistakes
- ILASP program implementation is faulty; this can also mean a few different things
  - **background knowledge** has faulty definitions
  - **examples** may not be constructed correctly
  - **hypothesis space** may not be constructed correctly

As with the actual cycle part, we need to consider doing things in parallel in post-processing. We may deal with hundreds of thousands of architectures, each of which should be labeled given the hypotheses we have. But it's quite obvious that, because at this point the hypotheses are constant, every architecture can be classified completely independently from the others, so we can indeed do this concurrently.

Finally, once all the initial architectures have been classified, the user can inspect them. If they see any inconsistencies, should they just redo the classification process (start the tool from scratch), or should the tool only terminate after the user has given their OK? During the cycle, the user classifies a set of architectures, but those labels should only be stored temporarily, otherwise they will just take up unnecessary space on the user's disk until manually deleted. This means that the architectures' labels should be deleted after execution ends, and so if the tool is run again with the same file, it will really start from scratch, with no initial hypotheses (other than the empty ones) or initial labeled architectures.

This suggests that the tool should wait for the user to give their OK. Another way to do this is as a "sanity check" for the user: continue classification and if all the new labels provided by the user coincide with the ones the current hypotheses would have given (called **predicted labels**), let the user know this (without redundantly rerunning ILASP). They may decide that the current hypotheses are, indeed, good enough and use them for classification, then terminate.

Given the independence of the three stages of the tool (pre-processing, learning cycle and post-processing), it was clear from the start that a good practice would be to split their implementation accordingly. The idea was to have a module for each of these stages (along with some potentially helper modules, e.g. utils), and call the relevant functions from a main script. This makes the tool more readable and extensible.

5.1 Prolog files with architecture descriptions

Before we look at the actual tool's structure, it's best to analyze what the tool has to work with: Prolog files describing a set of given architectures. For performance and safety reasons, the following assumptions are made with regards to the provided file:

1. It is a syntactically correct Prolog file.
2. Each model is defined with predicate: $model(model\_id)$.
3. Each of that model's nodes, ports and edges are defined *before* any other new model is defined and *after* its $model\_id$ is defined.
4. Each of $model\_id$'s nodes (components) are defined with predicate: $node(model\_id, node\_id)$.
5. Each of $node\_id$'s ports are defined with predicate: $port(node\_id, port\_id, group\_no, direction, type)$.
6. Each of $model\_id$'s edges are defined with predicate: $edge(model\_id, edge\_id, port\_id\_1, port\_id\_2)$.
7. No other predicates appear in the file.

Rule #3 may be a bit confusing; looking at the definitions of the predicates $node$, $port$ and $edge$, they are all "tied" to the model they belong to, so their position in the file should not matter, so they shouldn't have to be right next to where their $model\_id$ is defined. However, this rule is enforced for performance and readability reasons.

Firstly, if these predicates were allowed to appear anywhere in the file, an $O(n)$ search would have to be done for *each* such predicate (so basically for the vast majority of the predicates in the file; we consider n to be the number of architectures). With the rule enforced, this search becomes pretty much $O(1)$ (from the position of the model's id), since the string-sizes of the architecture descriptions are relatively small. Since these searches must appear in some outer loops (e.g. when iterating through all n given models), they will raise the overall complexity by one order of magnitude to potentially $O(n^2)$ or more, which for hundreds of thousands of architectures becomes very expensive.

Secondly, these architectures might require human inspection by the user anyway, so it's bad practice in the first place not to make them human-readable by not placing the models "in order". Since these models are likely generated, having the generation script place them in order is likely not going to be a burden for the user; this was the default format we received from Siemens, so this requirement was already fulfilled. All other assumptions presented above are also based on and in accordance with the datasets received from Siemens.

All the other rules present the format of the predicates the tool pre-processor expects. One thing to note is that these formats contain some extra information not included in the ILASP example contexts from the passive learning task from before (e.g. port type, port group number, direction, etc.). They are, however, necessary for differentiating types of architectures. For example, two architectures with the same components, ports and edges should not be considered "the same" if some port groups are defined in a different way in the two architectures. This proves that port groups are relevant data for *defining* architectures, even though they are not relevant for *classifying* them. More on this is covered in the pre-processing section later on.

Lastly, we present a sample **valid** model definition of an architecture:

```prolog
model(1).
node(1,n1).
port(n1,pr_2_0,-1,inout,rotational).
node(1,n2).
port(n2,pe_5_0,1,inout,electrical).
port(n2,pe_6_0,1,inout,electrical).
node(1,n3).
...
edge(1,c1,pr_2_0,ra_0).
edge(1,c2,pe_5_0,pe_9_0).
edge(1,c3,pr_10_0,fa_0).
```

Listing 5.1: Valid architecture definition in Prolog

It's worth noting that we don't actually have the component types defined at this point; the tool will not know from the given file which node is the vehicle, which is a gearset, and so on. It might not be relevant data for the user, but the diagrams should likely contain this information, so the pre-processor will have to identify **all** different types of components in the given file and either assign names to them itself or prompt the user to do so.

The tool will be able to do this process because, as can be observed in the listing above, although no component (node) is called a "battery" specifically, the node $n2$, along with its ports, contains all the information in order to be uniquely and unquestionably identified as a battery component.

Lastly, the port ids from the listing above only **happen** to contain relevant data about them (e.g. *pe* means the port is electrical, *pr* means it is rotational, *pr_14* means it is part of a gearset in this case), but in general the port ids will be completely random (usually *p* followed by a unique number, e.g. $p12345$), so no assumptions should be made about the port names in terms of what relevant information they may contain about the port itself; they shall be considered a unique identifier and nothing more.

5.2 Entry point

Moving on to the implementation of the tool, we begin with its entry point. The *main.py* file is a relatively short file, since it should not do much computation; its sole role is to be the entry point of the tool and call the relevant functions from the other modules.

5.2.1 How to run the script

Before we can go into more details about the contents of this file, its worth noting that, in order to run this script with a given Prolog file containing the architecture descriptions, we would call the following in the command line (from the directory containing main.py, for simplicity):

```
$ python ./main.py path/to/architectures/file/architecturesFile.pl
```

Listing 5.2: Calling the main python script

Given that a user who wants to use the tool should not need to call python with the entry point *main.py* and its path every time, those can be collapsed in one single script that can only be called with the desired architectures file. Here is one way to do that:

```
#!/bin/sh
INPUTPATH=$(readlink -f $1)
cd $(dirname "$0")
python ./modules/main.py "$INPUTPATH"
```

Listing 5.3: classilasp script

We save this in a script file called **classilasp** (combining the tool's **class**ification purpose and its underlying usage of **ILASP**), and save all the modules, including the *main.py* script file, in a directory called *modules* (having *classilasp* and *modules* in the same parent directory). Then the script from above first computes the absolute path of the given Prolog file, then moves (*cd*'s) in *classilasp*'s parent directory, and finally calls the required python command, now knowing the paths of both *main.py* and the given Prolog file, no matter where the user's working directory was when they called the *classilasp* script.

Now the user can add the path to *classilasp* to the PATH variable (ideally permanently, by altering *~/.bashrc*) and call:

```
$ classilasp path/to/architectures/file/architecturesFile.pl
```

Listing 5.4: Calling classilasp

5.2.2 main.py structure

We can now have a look at what the main file actually contains. First, this is the script that is run from it:

```
if __name__ == "__main__":
    try:
        main()
    except KeyboardInterrupt:
        print('\nInterrupted from keyboard!\n')
    except Exception:
        traceback.print_exc()
    finally:
        utils.deleteTempDataAndExit()
```

Listing 5.5: Main script

We see that this script runs a main function, which we'll present shortly, but it's important to note that this is done within a try-catch statement. The reason for this is quite simple: each execution will generate temporary files (containing ILASP programs to be run, stored labeled data, diagram image files, etc.) which have to be stored in a temporary directory to be deleted upon termination. If execution ends unexpectedly, or via a keyboard interrupt caused by the user, that temporary data should still be deleted, otherwise the user's disk will be left with a lot of unnecessary data, which is not the behaviour we want. That is why the temp data deletion is included in the **finally** bit of the try-catch statement.

Now let's have a look at the main function called in the script above:

```
def main():
    preProcessingFunc()
    while True:
        processingCycleFunc()
        postProcessingFunc()
        ...OTHER SMALL USER INTERACTIONS...
```

Listing 5.6: Main function

As expected, this function is only responsible with calling the pre-processing, cycle and post-processing functions, as well as some user interaction. These are still inside the main file, as they do some "general" work, like setting up the temp directory or asking for certain inputs from the user regarding certain global settings, but their main purpose is to call the relevant functions from their corresponding modules.

Note how the main learning cycle function and the post-processing one are in a "while **True**" statement; this is to allow the user to go back to improving the current hypotheses in case the results produced by post-processing are not satisfactory. Obviously, if/when they are, we **break** in order to exit this statement.

5.3 State module

Before doing an in-depth analysis of the main modules of the tool, we shall present a smaller one that was used mainly for ease of implementation: the state. It's worth noting that the scope of most variables of the tool is generally at most a module function, or usually as small as a simple loop. But there are some variables that are global to the tool. Instead of making all the modules have their own inner state containing all their variables and pass them around in functions with increasingly many arguments, it became clear that these should instead be extracted into one state module and have all other modules stateless, loosely based on the philosophy of the Redux library [8].

This approach makes it much easier to reason about how the tool works and generally the whereabouts of the important global information. The state should, however, not be abused and only these global variables should go in there, like the temporary directory path, the input file path, the current hypotheses, the whereabouts of the labeled data, etc. Though this has not been an issue with the tool implementation (as of writing this report, at least), concurrency issues should also be considered when handling the state.

It should be noted that thread-safety is not handled from within the state module, so any accesses from different threads (that we indeed sometimes have, as we'll see) should ensure this kind of safety themselves instead (a simple lock has worked flawlessly thus far).

The state module contains two simple functions: *get* and *set*, which are self-explanatory. This is what the state module looks like:

```
state = {
    'inputFilePath': '',
    'tempDirPath': '',
    ...,
    PROPERTY: DEFAULT_VALUE
}

def set(key, value):
    state[key] = value

def get(key):
    try:
        return state[key]
    except KeyError:
        return None
```

Listing 5.7: State module

As can be seen, the state object simply maps properties to values, having default values initially set for each of them.

5.4 Pre-processing

Now we can discuss one of the most important modules of the tool. The pre-processor has to "make sense" of the data presented to it in an input Prolog file. That data is considered to be correct (as per the assumptions presented in section 5.1) and should be fully parsed and processed before providing an output to the learning cycle. The question then is: what should that output be? We've seen what a model representation looks like (also in section 5.1) and that this is very similar to what we need to provide as example contexts in the ILASP learning tasks.

So can we then parse out the unnecessary arguments from the $node$, $port$ and $edge$ predicates and then simply return that from the pre-processor? Not quite, as there are some other things to consider here.

First and foremost, as previously pointed out, we don't have the component types in the $node$ predicates, and as we've seen in the passive learning task, the classification hypotheses do depend on these types (e.g. they were defined in terms of the *vehicle*, *battery* components). More importantly, we don't even know what those types might be; we only have a set of architectures (which are virtually sets of components). This means that we have to go through the entire file, keep track of all the currently computed component types, and whenever we encounter a "new" one (defining when two component types are "equal" is another issue to consider), we ask the user what name should be assigned to that component (or assign one automatically if they don't care about component names), and then keep doing this until the entire file has been processed.

Secondly, in the learning cycle we need to have a strategy when choosing what architectures to give the user for manual classification. Thus, pre-processing is where we can set up everything we need for this so that we don't have to do any unrelated work in the learning cycle other than the "choosing" of a candidate model, however complicated that can get in itself.

These are the main tasks the pre-processor currently has to handle, and we can now have a look at how it does them.

Computing all component types

It should be quite clear that these component types will be used throughout most of the computations done by the tool, and so we can safely assess them as global variables and place them in the state. It should also be noted that these can be considered constants after pre-processing is completed and thus should not be altered in any way.

So before we can start computing all different types of components, we need to define when two component types are considered to be equal. This then boils down to defining when two port groups are considered to be equal (since they are the immediate children of components), which lastly depends on defining when two ports are equal. This hierarchy suggests we should create these structures as actual objects for which we can formally define equality. We do this as follows:

1. We define the inner-most objects, which are the Ports:

```python
class Port(object):
    def __init__(self, portId, portType, direction):
        self.portId    = portId.lower()
        self.portType  = portType
        self.direction = direction

    def __eq__(self, other):
        if isinstance(other, self.__class__):
            return  self.portType == other.portType
                and self.direction == other.direction
        return False

    def __ne__(self, other):
        return not self.__eq__(other)

   ...
```

Listing 5.8: Port class

Note that the only attributes relevant for the Port object are the port **id**, **type** and **direction**, and we use only the last when checking for Port object equality. There are some other inner methods defined in this class, but they're not relevant to this discussion.

It's also worth pointing out that we don't use the port **group number** to check for Port equality. This is because the PortGroup is an object that has Port objects as inner variables, and so we use Ports to define equality for PortGroups rather than the other way around.

2. We then define the above-mentioned PortGroup objects:

```python
class PortGroup(object):
    def __init__(self, groupId, ports):
        self.groupId = groupId
        self.ports   = ports

    def __eq__(self, other):
        if isinstance(other, self.__class__):
            return utils.checkListsHaveSameElements(self.ports, other.ports)
        return False

    ...
```

Listing 5.9: PortGroup class

So we can see from the constructor that the PortGroup object is fully defined by its **id** and the **ports** it contains. This is important because we can check for equality completely based on these ports: if two PortGroup objects contain the same list of Port objects (potentially in different orders; note that we check for Port object equality, not actual object references equality), then they are considered to be equal.

3. Finally, we can define a Component object in a similar way:

```python
class Component(object):
    def __init__(self, compId, groups, name=None):
        self.compId = compId.lower()
        self.groups = groups
        self.name   = name

    def __eq__(self, other):
        if isinstance(other, self.__class__):
            return utils.checkListsHaveSameElements(self.groups, other.groups)
        return False

    ...
```

Listing 5.10: Component class

This follows almost the exact same principle as the PortGroup definition, having a list of PortGroup objects instead of a list of Port objects. It also needs its id, and the *name* attribute is set to **None** unless explicitly set otherwise.

All these definitions can be placed in a separate module, since they will also be used throughout most tool modules; for now, this module is named *architectureClasses.py*.

It's worth noting that for equality we never consider the *id*s, so equality is always done on *types* (which can be seen more or less as the set of all object attributes except the *id*, or rather the set of those parameters that are *structurally* relevant in the graphs), which is what we need for our task of computing all different component types. Now we could stop defining objects at component-level (so no objects for edges and models as a whole), and we could simply store the model strings, along with the component types. This is all the information we need to move further to the processing cycle.

However, instead of always having to parse strings in order to understand what components we are dealing with in our models, it's much easier and convenient (and more sensible, really) to have these objects defined as well, and work with Model objects for the remaining parts of the tool's computation. That allows for better debugging, more efficient computations (by less repeated operations) and more readable code.

So we can have a quick look at the most important bits in these definitions:

```python
class Edge(object):
    def __init__(self, ports):
        self.ports = ports.lower()

    def getPortAId(self):
        strRepr = str(self.ports)
        i1 = strRepr.index('(')
        i2 = strRepr.index(',')
        return strRepr[i1 + 1 : i2]

    ...
```

Listing 5.11: Edge class

Note that all an Edge object contains is a string with the pair of port ids it connects, i.e. "$(portAId, portBId)$" (e.g. "$(p123, p789)$"), which explains the two methods to extract these port ids: *getPortAId* and *getPortBId*, only one of which is presented above since the other is almost identical. The reason we don't store actual Port objects instead is because we simply don't have any reason to; port ids are all the information an Edge object needs to know about, so having access to all the other attributes of those ports is unnecessary.

Having defined Edge objects, we now have all the information we need to define the Model class:

```python
class Model(object):
    def __init__(self, modelId, components, edges):
        self.modelId    = modelId
        self.components = components
        self.edges      = edges

    ...
```

Listing 5.12: Model class

Note that we don't define equality for Models; there's no good reason to do so (yet, at least). Since the data is generated with the tool Siemens currently has, the models are known to be different, even from an isomorphic point of view, so the tool doesn't (or at least shouldn't) have to check for any kind of equality for Models. If, for the general case, this becomes too strong of an assumption, we can always go back to this and indeed define equality to mean "isomorphic", so that we don't give the user similar models to classify, which would not really improve the current hypotheses.

One last observation is that we haven't included the entire class definitions, as some methods were not relevant to the points discussed here (this was emphasized by the use of "..." in the code listings). One such method is probably worth mentioning: *getUsedComponents* in the Model class. In general, it might be the case that model descriptions in the Prolog files contain *all* possible components and only draw edges between *some* of them. As a result, though the components might still be relevant to their models, when presenting the user with a model diagram, we only want to show the relevant components, i.e. those that are actually part of the model graph via edges, which is where this function becomes useful.

Having all these definitions, we can write the outline of the algorithm (in pseudocode) for computing all the Model objects from the Prolog input file, while computing all the different types of components in the file. Note that whether these components should be named manually by user input or automatically by name generations depends on the user's preference, stored in the boolean $nameComponents$ (**True** if user prefers manual naming, **False** otherwise).

```
# Separate input file string into list of strings, one per model
# (with all its defining predicates, i.e. model, node, port, edge)
modelStrings = getModelStringsFromFile(inputFilePath)

# componentTypes is the list of different components to be computed.
# Initially, this list is empty
componentTypes = []

# modelObjs will contain the Model object equivalents for all string
# representations in modelStrings
modelObjs = []

# Iterate through all model strings from input file
for each modelStr in modelStrings
    # Compute all Port, Edge, PortGroup and Component objects from modelStr
    ports      = computePorts(modelStr)
    edges      = computeEdges(modelStr)
    portGroups = computePortGroups(modelStr, ports)
    components = computeComponents(modelStr, portGroups)

    # Iterate through all computed components in current model
    for each component in components
        # See if current component already exists in componentTypes
        compType = find(component, componentTypes)

        if compType not null
            # If it does exist, simply use the name already assigned
            component.name = compType.name
        else
            # If not, add this new component to componentTypes
            # and generate a name for it (either by asking
            # the user, or automatically, based on the value of
            # nameComponents)
            component.name = generateName(component, nameComponents)
            componentTypes.add(component)

    # Once this is done, we can define the Model object, and add it to
    # modelObjs
    model = computeModel(modelStr, components, edges)
    modelObjs.add(model)
```

Listing 5.13: Computing Model objects and Component types pseudocode

At the end of this computation, we will have computed the Model objects for the model representations in the input file (including the names for their inner components, assigned also during the computation from above). Note that, for simplicity, we had $componentTypes$ as a list in the scope of the computation, although, as said before, these belong in the state and so some state accesses are required to modify and update them during the loop.

It’s also worth showing how the component naming process works. In case the user decides that they don’t want to manually name the components themselves, the tool will have to assign the names instead. This is done simply by keeping a counter and whenever a new component arises, use the counter to name it and increment the counter:

```
counter = 1
while pre-processing
        if new component detected
        name = "type" + counter
        counter += 1
        assign name to component
```

Listing 5.14: Automatically naming components

Now the more difficult task is when the user does want to name the components. In that case, the question that arises is: what’s the best format to display the component to the user so that they can easily see everything it contains and be able to identify and name it? A diagram again seemed like the right way to go, so we came across the Python library **graphviz** [3], which generates very nice and clear diagrams, with a simple, intuitive API.

So basically the naming process is the same as in the automated version, only this time, whenever a new component is detected, we generate a graph diagram from it using *graphviz* (saved in the temp directory as an image file), display it on the screen and wait for the user to decide on its name. We also obviously no longer need the counter.

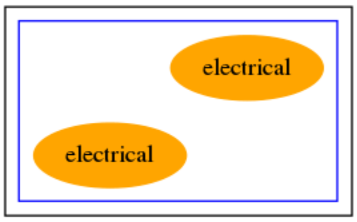


Figure 5.2: Battery component diagram

In Figure 5.2 we see what the user would be shown when the battery component was detected. The user would then have to inspect this diagram and indeed name it "battery". But before that, we should explain what everything in this diagram represents:

- all the round-shaped, labeled dots are the ports; the labels refer to their type, while their colour reveals their direction: red for "out", green for "in" and orange for "inout"
- the ports are organized together via blue boxes; these represent the port groups
- the port groups are then surrounded by the outer black box that represents the component delimitation

This is the entire information that was provided in the input file about this component, so the user should be able to recognize and name it successfully. Here is another, more complex, diagram of an unnamed component, to better highlight the notions above:

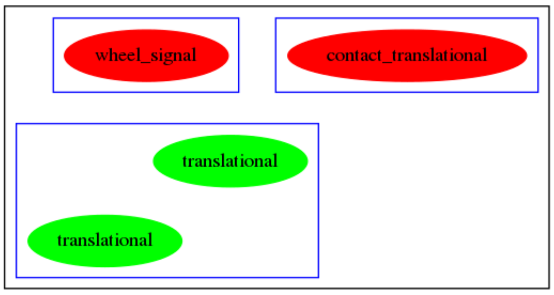


Figure 5.3: A more complex component diagram

MeanShift Clustering

This is where we use the MeanShift algorithm presented in the Background chapter. Namely, we make use of the **sklearn.cluster** module from the **scikit-learn** framework ([31]) for this. It simply takes our models' component-frequency vectors as input and outputs the clusters. Although we will present an algorithm to continuously update these clusters in order to improve their efficacy, MeanShift's output is sufficient as the *initial* clusters.

Random sampling from the dataset

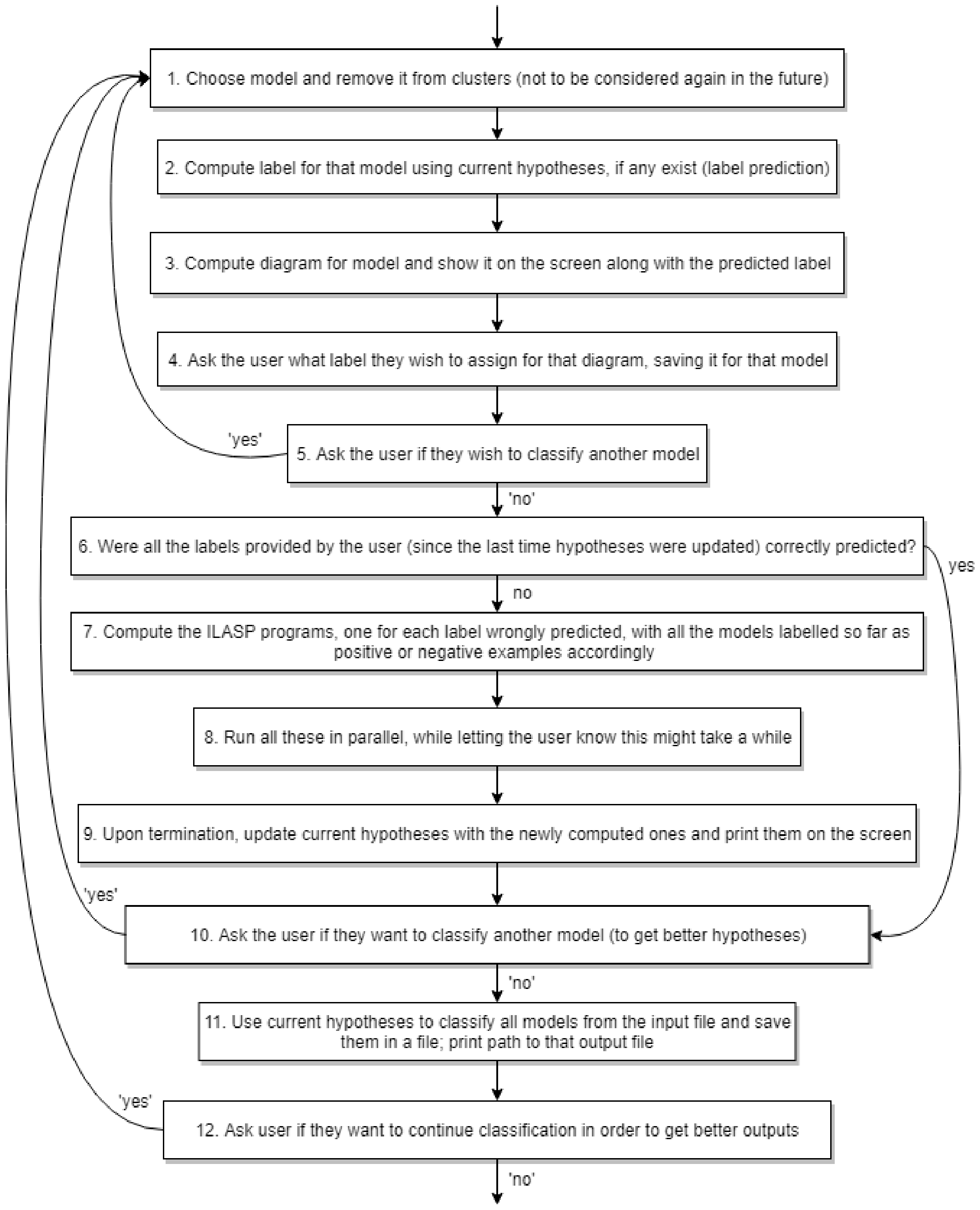
Before presenting the learning cycle, a small observation needs to be made. One simple optimization that can be done at the beginning of pre-processing is random sampling. There's no reason to store hundreds of thousands of Model objects when only a handful of them will actually get to the user in order to be labeled. Of course, we shouldn't make assumptions about how many models the user wants to classify (restricting this number to, say, 10 or 20 would be too much), but it's safe to assume they won't go as far as labeling, say, a thousand models. This upper boundary is stored in a constant called MAX_RELEVANT_MODELS in a module storing all global constants, and thus the user may configure it to their own preference (e.g. if they feel 1000 is still too small for them).

So if the dataset contains more than MAX_RELEVANT_MODELS, we do random sampling so that we are left with exactly MAX_RELEVANT_MODELS models to be considered, and the rest will be discarded; if it doesn't exceed that value, we keep all models. Since sampling is done completely at random, the overall distribution of labels over the models we are left with should stay roughly the same, so no information is lost in that sense.

There is another reason to do random sampling, other than practicality: time complexity of MeanShift. According to [6] and [7], MeanShift is not exactly scalable, with a time complexity of roughly $O(KN^2)$, where K is the number of iterations until the centroids stabilize, and N is the number of samples. This means that, even though K is in general very small, random sampling down to some smaller number of models from a few hundreds of thousands is not exactly a choice, but a necessity.

All in all, pre-processing is quite fast (for about 170.000 architectures with 10 types of components it takes around 4-5 seconds, not counting time spent during component naming, in case user input is required) and also memory-efficient (even when the input file contains millions of architectures, it never stores more than MAX_RELEVANT_MODELS Model objects - currently 1000 - and all other large resources are garbage collected after pre-processing is done). These aspects are analyzed in much more detail in the Evaluation chapter, and we can therefore move on to the next important part of the tool: the learning cycle.

5.5 Learning cycle



1. Choose model and remove it from clusters (not to be considered again in the future)
2. Compute label for that model using current hypotheses, if any exist (label prediction)
3. Compute diagram for model and show it on the screen along with the predicted label
4. Ask the user what label they wish to assign for that diagram, saving it for that model
'yes'
5. Ask the user if they wish to classify another model
'no'
6. Were all the labels provided by the user (since the last time hypotheses were updated) correctly predicted?
yes
no
7. Compute the ILASP programs, one for each label wrongly predicted, with all the models labelled so far as positive or negative examples accordingly
8. Run all these in parallel, while letting the user know this might take a while
9. Upon termination, update current hypotheses with the newly computed ones and print them on the screen
'yes'
10. Ask the user if they want to classify another model (to get better hypotheses)
'no'
11. Use current hypotheses to classify all models from the input file and save them in a file; print path to that output file
'yes'
12. Ask user if they want to continue classification in order to get better outputs
'no'

Figure 5.4: Learning cycle flowchart

In Figure 5.4 we present an overview of what the learning cycle looks like. Note that this is, in principle, based on the high-level diagram in chapter 2, but with much more detail. In the next few pages we shall have a closer look at how each of the steps works.

5.5.1 Structure

Step 1

The entry point is step 1, which requires a fresh model to be generated. The mapping of cluster ids to the lists of models belonging to the corresponding clusters was constructed in pre-processing and should look like this:

```
{
  '0' : [model_3, model_7, ...],
  '1' : [model_1, model_4, ...],
    ...
  'k' : [model_2, model_1857, ...]
}
```

Listing 5.15: Clusters mapping

Note that the order of models in each cluster has already been randomized in pre-processing, so that we don't have to select a random model from a given cluster at this stage; popping the last model in that cluster is already random. The reason why we have to choose **random** models from clusters (even after the clusters were chosen at random) is that the order in which the models were given initially in the input file may contain some sort of bias (e.g. first half of the models may have, in general, less components than the second half; this means that, if we don't generate random models from clusters and instead choose them in the initial order, it may take a lot of time until we get to the models with more components, from the second half).

So in order to generate a random model, we generate a random *key* for the clusters mapping (same as choosing one of the clusters at random). Then we go to the list corresponding to that *key* and pop the last model in that list. This will be the **chosen model** to be used in the next steps. Before we move on to those, a couple of things to consider here:

- because MeanShift does not require the number of clusters to output, we are guaranteed that each cluster it outputs is non-empty; this means that we are guaranteed that the first time we enter step 1, we will have a model to select, no matter what cluster we choose from

- we *pop* the chosen model from its cluster; this way we make sure that that model cannot be chosen again for consideration in the future, since it will no longer be in any of the clusters

- because we pop the chosen model, its cluster may become empty afterwards; if that becomes the case, we delete the cluster from the mapping, so that it won't be up for consideration in the future

- because we delete clusters as they become empty, it may be the case that we are left with no clusters. This is equivalent to running out of models for classification; this is **extremely** unlikely to happen in general. Datasets are expected to be very large, given that they require automated classification. Even after restricting these to a maximum of 1000 to be considered (at least at the moment, as explained in pre-processing), the user is not expected to ever have to classify this many models before getting valid hypotheses for their labels.

  Even so, we have placed a safety check in the code in case this happens: if we completely run out of models, we must alert the user, update the hypotheses to include all newly classified models and use them for classifying all the initial models **no matter what** (the specifics of how these operations are performed are explained in the next steps), and exit. This extra step isn't included in the flowchart because it is a small safety check, and the flowchart is supposed to present the *general* overview of the cycle.

Step 2

Then we have to compute the label(s) entailed by the hypotheses we currently have, i.e. the predicted labels. Note that if all hypotheses are so far empty, no predicted labels exist. We do this for both the user and the tool itself:

- the user: displaying the predicted labels along with models can constitute a sanity check after a while; if the tool consistently predicts labels correctly after a while, that's a good indication that the current hypotheses are good enough and the user can ask the tool to use them for classifying all the initial data. Also, predicted labels may also help the user perform the manual classification more quickly.

- the tool: in general, if a label is correctly predicted, this means two things. Firstly, it means that that label's hypothesis is correct, as it rightly entailed that label from the model description. Secondly, it means that all other labels' hypotheses are also correct, since they didn't entail their own labels for that model. This means that we don't need to update any hypothesis, for this model at least (they may have incorrectly predicted labels for other models and so they will have to be recomputed because of those)

The way to do this is quite simple: we generate the ASP program containing the model description, as well as the hypotheses; after running *clingo* with this program, we simply look for the labels in the answer set.

Step 3

This step comes as preparation for the manual classification of the model by the user. Since this classification is done best using visuals, a diagram representation of the model is needed. We use for this the same library we used for pre-processing (when showing components to be identified and named by the user): **graphviz**. As said previously, we also display the predicted label(s) for that model, in case they exist. Here is an example of such a diagram:

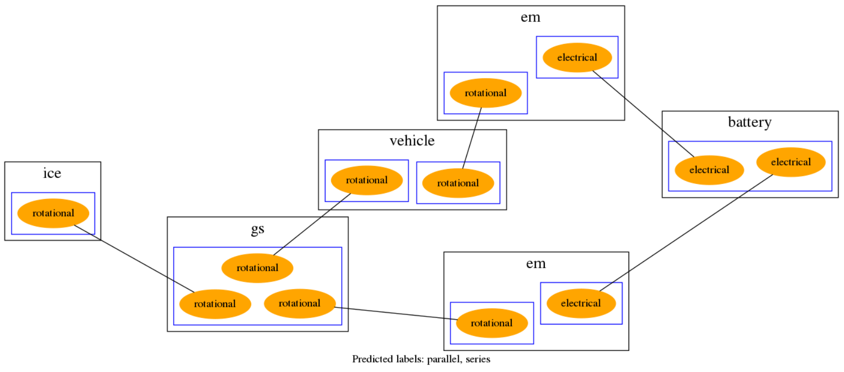


Figure 5.5: Model diagram with predicted labels

In this case, the model happens to be predicted as both *parallel* and *series*, so it is definitely at least partially mislabeled. The user can use these predictions to hopefully see more quickly what the right label actually is.

Step 4

Based on the diagram generated in the previous step, the user should now be able to analyze it and know what label corresponds to it (as all the information from the input file is captured by the diagram). Hence, we ask the user what that label is and store it along with the model.

Now, it's important to note that the only reason why we need the classified models is to use them as positive or negative examples in the ILASP programs to be run in order to obtain hypotheses for the labels. So there's no reason to store the actual Model objects, along with their labels, and compute the examples as strings every single time we want to run ILASP. Instead, because we need to compute one hypothesis for each label, we will need as many ILASP programs as there are labels.

It is likely possible to instead have one big ILASP program that could compute all hypotheses in one go, but there simply is no reason to take this approach. This would not only recompute **all** hypotheses every time (instead of recomputing only the ones that wrongly predicted labels so far), but it would most likely be slower because of the obviously larger hypothesis space. Another reason is that it's also a bit easier to reason about completely separate tasks and what they are supposed to compute, rather than one big program trying to compute several tasks in one go.

So having established that, we can now separate the work even further. For each label, we can have a file containing only the examples corresponding to that label. That way, every time a new model is classified, we simply add its string equivalent (as an example) to all those files. Also, this model would obviously be a positive example in the file corresponding to the label the user selected, and a negative example in all other files.

For example, if we had the three labels, a, b and c, and a new model that the user labeled as b, then the corresponding files with examples for the three labels would now look like this:

```
...previous examples...

#pos({}, {a}, {
comp(n1,ice).
val(n1,1).
port(n1,pr_2_0).
...
edge(pr_2_0,pr_10_0).
...
}).
```

(a) Label a examples file

```
...previous examples...

#pos({b}, {}, {
comp(n1,ice).
val(n1,1).
port(n1,pr_2_0).
...
edge(pr_2_0,pr_10_0).
...
}).
```

(b) Label b examples file

```
...previous examples...

#pos({}, {c}, {
comp(n1,ice).
val(n1,1).
port(n1,pr_2_0).
...
edge(pr_2_0,pr_10_0).
...
}).
```

(c) Label c examples file

Figure 5.6: Examples files for each label

What the new examples say, respectively, is that:

- the new model is not of label a
- the new model is of label b
- the new model is not of label c

We haven't included the entire model descriptions in the example contexts, but they are obviously the same in all three files, the only differences must only be in the first two arguments of those examples. Note, also, that the example files are specific to every execution of the tool, so they must go in the temp directory, in order to be deleted at the end of execution.

Lastly, since the model the user labeled is now stored as an example in these files, there's no need to store the actual Model object anymore, nor its label, so by moving on to the scope of the next step, they will simply be garbage collected.

Step 5

This step is simply to enforce the cyclic pattern of the overall algorithm: we ask the user if they wish to classify another model or not. If they say 'yes', then we can simply go back to Step 1 and provide them with another model to be labeled. If not, we move on to the next step.

Step 6

In this step, we check how accurate the current hypotheses were in predicting the labels the user provided since the last time these hypotheses were updated. Note that by saying that a hypothesis *positively* (*negatively*) predicts a model we mean that that hypothesis entails (doesn't entail) the label corresponding to that hypothesis for that model. For example, the hypothesis for label a positively predicts a model if it entails the label a for that model. It negatively predicts that model it doesn't entail label a. We may alternate the actual phrasing (e.g. model predicted *as a negative* instead of *negatively*), but we always mean the same thing described in this paragraph.

We touched on this earlier, but here are some of the slightly different cases to be considered:

- one model was labeled a and predicted as b. This means that both the a and b hypotheses were wrong, but for slightly different reasons: the a hypothesis incorrectly predicted the model to be a negative (since it wasn't predicted to be an a model), while the b hypothesis incorrectly predicted it as a positive. All other hypotheses correctly predicted the model as a negative. This means only the hypotheses for a and b need to be recomputed to cover this model.

- one model was labeled a and no predicted label existed. This is trivial: the a hypothesis is the only one that shouldn't have predicted it negatively, so it needs to be recomputed; all other hypotheses can stay the same.

- one model was labeled a and predicted as b and c. Note that this is indeed possible, as explained before, since the hypotheses are only mutually exclusive on the examples currently available in the ILASP programs that compute them (every example is positive for one label and one label only), but not mutually exclusive in general, so any new incoming model can potentially be predicted to be any number of labels. The reasoning is very similar to the first point from above, and we obviously need to recompute the hypotheses for a, b and c; no other hypotheses need to be recomputed.

Note that we don't recompute these hypotheses after every newly classified model, as these can take quite long. That's why we only recompute them when the user decides to do so, as the previous step emphasizes.

Using the points described above, we can keep track (in the state) of the hypotheses that definitely need recomputing. The whole purpose of this step is the check whether there are any hypotheses that do need recomputing. The reason for this obvious: if there are no such hypotheses, then no ILASP tasks have to be started, and we can directly move on to step 10 to ask the user what they want to do at this point (having let them know that the current hypotheses have been 100% accurate with all the new classified models).

If any hypotheses were inaccurate and need recomputing, we simply move on to the next step.

Step 7

Given the labels for which we need to (re)compute hypotheses, we have to actually generate the ILASP programs to be run. These programs must have their three important parts: **background knowledge**, **examples** and **mode bias**.

The background knowledge should be, as its name suggests, general enough to be constant for all labels; any further, more specific, knowledge can go in the example contexts. So we can simply recycle the background knowledge from the passive learning task, as that contained only the very general predicates ILASP should uses:

```
compV(C) :- comp(C, _).
```

```
edge(P1, P2) :- edge(P2, P1).
are_connected(C1, C2) :- port(C1, P1), port(C2, P2), edge(P1, P2).
```

Listing 5.16: Background knowledge file

Additionally, this should also include predicates such as *basic_path*, *direct_path* and all other general substructures ILASP should know to look for in the given architectures. We store the background knowledge in a separate file - *background.las* - and because this is information necessary in this exact form for every execution of the tool, this file should not go in the temp directory, but instead should be persistent, along with the other Python modules.

The examples are, as discussed in step 4, stored in temporary files, one for each label, in the form presented in Figure 5.6.

The mode bias is not quite as general as the background knowledge. First of all, each hypothesis space should look for a hypothesis "explaining" one label, e.g. for the hybridization case, one hypothesis should "explain" *parallel*, another should "explain" *series*, and so on, so they cannot be the exact same. One way to do this in a generic way is to have a persistent bias file, like the following:

```
1 #modeh($$LABEL$$).
2
3 #modeb(1, direct_path(var(compV), var(compV), var(compV))).
4 #modeb(3, comp(var(compV), const(compC))).
5
6 #modeh(invented_pred, (non_recursive)).
7 #modeb(invented_pred).
8
9 $$CONSTANTS$$
```

Listing 5.17: Generic mode bias file

This is still general enough and always contains the same predicates, *direct_path*, *comp* and an invented predicate *invented_pred*, to build the hypothesis space. However, we also have *#modeh($$LABEL$$)* in line 1, where the string "*$$LABEL$$*" can be easily replaced with the actual label for each ILASP program, in order to get the rules with the required label as their head in the hypothesis space. Similarly, line 9 contains a place-holder for the program constants; these can be component types (e.g. vehicle, gearset) and are specific to the input file for each tool execution, which is why they aren't constant in general.

The ILASP programs are simply the three parts presented above, concatenated as strings, for each label that needs their hypothesis recomputed.

Step 8

Once we have the ILASP programs, we can run them to get the new hypotheses. Note that these are completely independent from each other at this point; this means we can run them in parallel. Given that we have to run actual ILASP commands, these will need their own processes, and the way to do this in Python is as follows:

```
def runILASPCommands(labelsToUpdateHypotheses):
        ...
        outputs = {}
        lock = Lock()
        threads = list()

        for label in labelsToUpdateHypotheses:
                threads.append(Thread(target=runILASPCMDInThread,
 9                                  args=(..., label, outputs, lock)))
10 
11         for thread in threads:
12                 thread.start()
13 
14         for thread in threads:
15                 thread.join()
16 
17         utils.updateHypotheses(outputs)
18         state.get('hypothesesToUpdate').clear()
```
Listing 5.18: runILASPCommands function

Again, we only include the relevant parts of the function. We obviously need to store the **output** from these ILASP programs, for each label, so we use a map for that. Since this output is handled from multiple threads, it's best to also use a lock when handling it. In lines 7-9 we initialize the commands that need to be run, passing all the arguments they need (including their label, the outputs map and the lock). Then we run them and block the current thread on these new threads, as we now need to wait for their output. Note that, before running this function, the user has been notified that the hypotheses are about to be computed and this might indeed take a while.

Step 9

Once all tasks have finished computation, the current thread is unblocked and it can update the hypotheses in the state with the new ones, and clear the list of labels whose hypotheses need to be recomputed. The new hypotheses are also printed on the screen for the user to analyze. The hypotheses should get better and better as the learning process goes on, so the structure of the hypotheses will start resembling more and more what the user intuitively uses when manually classifying given architectures. At some point, they can become certain that using the latest hypotheses will produce results very similar to what they would have done on their own, manually.

Step 10

Regardless of whether hypotheses had to be recomputed or not (based on their label predicting accuracy), the user will get to this step knowing the current hypotheses that they have. With them in mind, they will get asked how they want to proceed: do they want to classify some other models so that they can obtain better hypotheses or are they happy with the ones they currently have and wish to use them to classify all the models in the input file? If they choose the former, they will go back to step 1; otherwise, they move on to step 11.

Step 11

This step uses the current hypotheses to classify all the models in the input file. This is considered the *post-processing* stage of the tool and is discussed in more detail in section 5.6. The important thing is that the labels are saved in a new file in the same directory as the input file (with the same name, plus a random short string to avoid name conflicts), and we can print the absolute path to this new file for the user to inspect.

Step 12

In general, it is unlikely at this stage that the user would continue classification. They would have to inspect architectures selected more or less at random from the potentially hundreds of thousands in the input file, and see if there are inconsistencies. But this is what the tool does automatically for them during the learning cycle, and the assumption is that the user would decide on using the current hypotheses for classification only after they consistently produce correct label predictions.

Regardless, we use this extra step in order to give the user the freedom to continue classification if they choose they would like to do so; if they end execution now, all temporary data is deleted, and that includes all the models that they classified during the learning cycle, so it's best to let the user decide if they wish to exit now instead of exiting automatically.

This is also a matter of testing too. Obviously, while testing the tool, we work with pre-classified data, so it is at this stage that we check how accurate classification was and decide whether we wish to improve the hypotheses or not. That way, it is easier for us to test how the hypotheses are getting increasingly better without having to run the tool more than once for a given dataset.

So finally, we ask the user if they wish to improve the hypotheses or end execution. Choosing the former sends them back to Step 1, while choosing the latter does the obvious and ends execution (when we can safely delete all temporary data).

5.6 Post-processing

Since pre-processing was the stage where the tool prepared the given data in order to use it as input for the learning cycle, post-processing is the part where the output of that cycle (i.e. the hypotheses) to further process the initial models and find their corresponding labels. This is, in fact, using the exact same technique described in step 2; the only difference is that, in that step we were *predicting* labels for models before getting the actual label from the user, whereas now we are simply *deriving* the final labels. The process behind is exactly the same, only the purpose differs.

So as before, we need to generate the ASP program (including the model description predicates, as well as the hypotheses), run that program using *clingo* and derive the labels entailed in the answer set in a special binary predicate $label(\_, \_)$, where the first argument is the model id, and the second - the label. We obviously don't need in the output all the predicates in the answer set, so we can only display this predicate by adding the following (where $/2$ says to only show the **binary** predicates $label$):

```
#show label/2.
```

Listing 5.19: Showing only labels in answer set

Once again, we can use concurrency to speed up the process. We do need a lock on the output file, but otherwise, the actual *clingo* computations can be done completely independently. Because these are actual processes that run in parallel, they tend to take a bit longer than one might expect; on their own, they are quite fast, but we have to now run these for **each** of the models in the input file. This means that we can no longer use random sampling or other optimization techniques used in pre-processing that brought the number of models to be processed from hundreds of thousands down to 1000.

Running one model per process, it takes, on average, about 0.1s per model, which is as expected (allowing for the extra steps Python does in terms of reading and writing from and to files, setting up the processes, etc, as opposed to running the *clingo* command directly from the command line). When we work with more than 170.000 models, however, the total computation time adds up to about five hours. This would be another reason why the user might not want to go through post-processing too often and instead make sure to check how well the hypotheses predict labels before using those for this final classification.

A different way to solve this issue would be to instead generate only one massive ASP program containing all the model descriptions and the hypotheses. Obviously, the hypotheses would have to change slightly (since simply entailing e.g. $parallel$ no longer means anything, as we would need to know **which model** entailed it specifically), but that's not even the main issue with this approach. The main problem is clearly the vast amount of resources this process would take up. By running tasks in parallel, we can restrict the maximum number of processes to be used, and thus the total memory they can use (assuming every process would require more or less the same amount), whereas running pretty much all tasks at the same time in one huge process would require a much, much larger amount.

However, there is a third way post-processing can be optimized, and it borrows ideas from both of these approaches. This optimization is indeed surprisingly powerful, bringing the computation time for the 170000 models from 5 hours down to just 2 minutes. So the **basis** of *classilasp* is indeed now in place and we can discuss details of further such optimizations and other features added on top of this basis in the next chapter.

Chapter 6

Further features and optimizations

Once the basic tool is functional and outputting valid, accurately classified data, we can build further features on top of that. Also, we can analyze what parts in its structure can be optimized in order to speed things up, either with simply "better" algorithms, or with more usage of the user's bias. All of these are discussed in this chapter, but here are the most important ideas:

- Although MeanShift clustering works quite well initially, we make use of the hypotheses generated by ILASP in order to continuously **recluster** the remaining models. The reasoning is clear enough: they generally offer more valuable information than the generic component-structure captured by Meanshift.
- We output **statistics/diagrams** showing the overall label distribution over the given models with the current hypotheses.
- We upgrade the pre-processor to allow for **pre-named model components**, so that the user doesn't have to name them in that case.
- We add the functionality to allow the user to (temporarily) **skip a model** they don't want to classify at the moment.
- We allow for and (up to a certain point) correct user mistakes, using ILASP's **noise** functionality.
- We formally define some more general **patterns**, apart from *direct_path*.
- We add a **querying** functionality. This refers to allowing the user to type in some constraints and then return models from the full dataset complying with those constraints, much like a database query.

Clearly, querying and classification work almost identically, since the hypotheses are the equivalent "constraints" from querying that are run for each model with *clingo*, so the overall computation time is expected to also be mostly identical. Obviously, queries are expected to be used quite often, unlike a full classification of the entire dataset. But because classification, in the form presented so far, takes several hours, one clear optimization to be analyzed and discussed in this chapter also becomes:

- A way to improve the classification process; as said before, keeping this computation in separate processes is still important from a space complexity point of view, but we specifically look at how to **remove** as much of the Python **multi-processing** setup **overhead** as possible.

With that said, let's have a closer look at each of these ideas and how they were approached and implemented.

6.1 Reclustering

As previously said in more detail, models in our random sample are stored in clusters, created in pre-processing. Using the MeanShift algorithm, models with similar compositions (in terms of number and types of components they contain) are placed in the same cluster. That does work well enough initially, but it obviously still clusters together **a lot** of models with different labels. In the case of the 170.000+ models, there are only 3 series architectures. Let's say we had a cluster for the $series$ label, and with the current hypotheses, that cluster contained 500 models (instead of just 3). Then selecting a model from this cluster would almost surely produce a non-$series$ model, and thus help improve the current $series$ hypothesis the next time the user asks for their recomputation.

So as emphasized before, the *ideal* clustering would coincide with the actual correct classes, i.e. have a cluster for each label and have **all** and **only** models of that label in that cluster. Obviously, the initial MeanShift clusters would in the vast majority of cases not coincide with those clusters, so we do need reclustering if we wish to get there.

It's important to remember that the whole purpose of our learning cycle is to learn the *ideal* hypotheses, i.e. those hypotheses that correctly classify every single model in the initial dataset. That means that those hypotheses can be used to construct the *ideal* clusters. So it makes sense to recluster based on the current hypotheses every time they are recomputed; since those clusters are a function of the hypotheses, when the hypotheses converge to the *ideal* hypotheses, so will the clusters converge to the *ideal* ones:

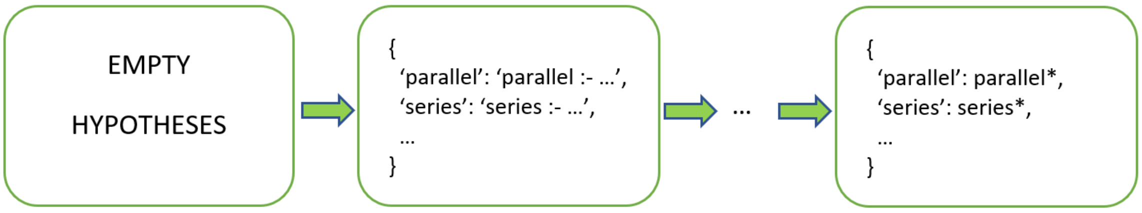


Figure 6.1: Hypotheses convergence towards the *ideal* ones

where $parallel^*$, $series^*$ are the *ideal* hypotheses for the corresponding labels, implies the corresponding convergence:

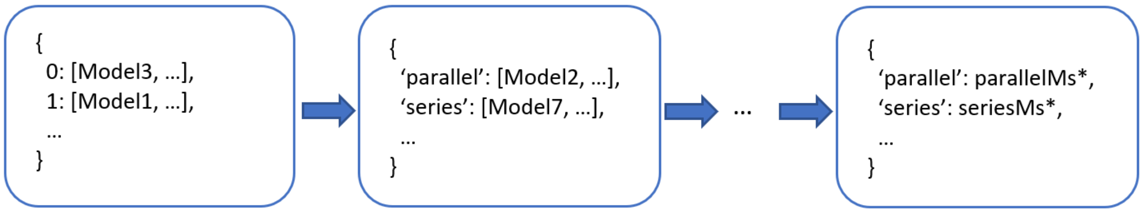


Figure 6.2: Clusters convergence towards the *ideal* ones

where $parallelMs^*$, $seriesMs^*$ are the *ideal* clusters of models for the corresponding labels.

Without going into much detail about implementation, as it is mostly intuitive, the main idea is that every time the user asks for the hypotheses to be (re)computed, the clusters are also updated accordingly. One very important aspect here is that this technique allows us for another strategy to speed up the learning process: assigning **weights** to each cluster. Previously, each cluster was equally likely to produce a model that could improve the hypotheses (i.e. with incorrect predicted labels), but with this approach, we have to emphasize again that the hypotheses are not, in general, mutually exclusive; each example is positive for exactly one label ILASP program, so the hypotheses we obtain are guaranteed to be mutually exclusive on those examples, but any further model can have multiple, or even no predicted labels.

That is, when computing the predicted labels corresponding to each model (in the random sample), we have two **special** clusters:

- **'No label'**: this contains all models with no predicted labels by the current hypotheses
- **'Multiple labels'**: this contains all models with more than one predicted label

Obviously, all other clusters will contain models with exactly one predicted label, namely the one corresponding to that cluster; we will refer to these clusters as the **normal** clusters from now on. It's important to note that, while some models in the normal clusters may be mislabeled, this becomes more and more unlikely (per model) after every recomputation of the hypotheses. However, every model in the special clusters is *guaranteed* to be mislabeled (either partially - if the right label is among the predicted ones, or entirely - otherwise).

This clearly suggests that the special clusters must have larger weights than the normal ones in order to speed up convergence towards the ideal hypotheses. It is expected that these two clusters will become smaller and smaller the closer we are to the ideal hypotheses, but they remain equally important. The obvious question that arises is: what weights do we give them? We don't want to exclusively choose from them, since some models in the normal clusters might still be mislabeled, potentially for a different reason than any model in the special clusters (so they can improve the hypotheses in a way the special clusters cannot).

Given their importance, we've decided to, in general, give each (of the two) special clusters a weight N, where N is the number of non-empty normal clusters (or 1 as default in the special case where $N = 0$). Additionally, every normal cluster gets a weight of 1. It should be mentioned that the clusters map (stored in the state) only stores non-empty clusters (as lists), including the special clusters. This means that, assuming that there is at least one remaining model in the clusters, we have the following cases:

1. $N > 0$ and both special clusters are non-empty. That means the 'No label' cluster has $\frac{N}{N+N+N\cdot 1} = 33\%$ chance of being chosen the next time the user asks for another model to classify. Similarly, the 'Multiple labels' cluster also has 33% chances, whereas every normal cluster has $\frac{33}{N}\%$ chances (e.g. 11% chances for each of 3 normal clusters, as in the hybridization case with normal clusters *parallel*, *series*, *series-parallel*). Overall, this means there is a 66% chance of choosing a special cluster which guarantees to improve the current hypotheses.

2. $N > 0$ and only one special cluster is non-empty, say the 'No label' one. Then, similarly to the previous case, that special cluster has a chance of $\frac{N}{N+N\cdot 1} = 50\%$ getting chosen, while every normal cluster has a chance of $\frac{50}{N}\%$ (e.g. 16.7% per cluster in the hybridization case). So each cluster has a larger probability of getting chosen in this case (compared to case 1), but the overall probability of obtaining a cluster (and thus a model) that is guaranteed to improve the hypotheses has decreased from 66% to 50%.

3. $N > 0$ and both special clusters are empty. This is identical to how the MeanShift clusters behaved, since in that case all clusters have equal weights; each cluster is normal and has a chance of $\frac{100}{N}\%$ to be chosen (33% in the hybridization case). Clearly, we have 0 probability of getting a model guaranteed to be mislabeled in this case.

4. $N = 0$ and both special clusters are non-empty. This, in a way, is also identical to the behaviour of the MeanShift clusters, since we again have two clusters with equal weights (of 1), it only so happens that they are both special. Since we have exactly two non-empty clusters, each has a 50% chance of getting chosen, and because both are special, we are 100% guaranteed to get a mislabeled model in this case.

5. $N = 0$ and only one special cluster is non-empty. This is trivial: there's only one cluster, so we are 100% guaranteed to choose the next model from it. Since it happens to be a special cluster, we are also guaranteed that that next model will come mislabeled.

Note that there isn't a sixth case where $N = 0$ and both special clusters are empty, because that implies there is no model left in the sample; however unlikely that is in general, it is dealt with separately as a special case within the cycle code, but it is not relevant to the discussion here.

With these new clusters computed after each recomputation of hypotheses, the rest of the implementation remains unchanged: choose randomly one (non-empty) cluster, now taking into account weights, then pop a model randomly from that cluster; if the cluster becomes empty, remove it from the clusters map.

6.2 Statistics and diagrams

In general, given a very large number of models in the initial dataset, the user will not know the *exact* number of models corresponding to each label; for testing purposes, we did have these exact labels for the 170.000 dataset and we were able to test the exact accuracy of the hypotheses we obtained.

However, the user will not have those exact labels (obviously, that's what they would like to obtain from our tool), but they might know some approximations. These might either be actual numbers, or simply proportions (e.g. they might expect "about two thirds of the models to be parallel" or "barely any to be series", as opposed to "102185/172767 are parallel" and "3 are series").

This is were statistics and diagrams produced by the tool may be vital: if these show numbers more or less agreeing with the user's expectations, that might be a good indicator that the current hypotheses are good enough and they can ask to terminate execution. There are two stages of the whole cycle where these statistics can be displayed and interpreted/used by the user:

- when new hypotheses have been computed. With the changes described in the previous section, this is where reclustering is done. That's in fact very useful, because reclustering actually computes the exact statistics that we need at this point: it computes the predicted labels for all remaining models in the clusters. If we store separately counters for the labels already provided by the user, we basically know at this stage **all** labels the current hypotheses entail for the random sample.

- when the entire dataset has been labeled. This is obviously the most important statistic, since it clearly shows the user how many models correspond to a certain label, how many models have no/multiple labels, etc. This is done quite easily with a few counters while doing the labeling of the dataset.

6.2.1 Expected values

Since the random sample is chosen specifically to represent the entire dataset while stile performing operations on it relatively quickly, for any value we obtain about it, that value can be seen as an **expectation** for the same value computed for the entire dataset. So, as said above, once reclustering is done, we know exactly how many models in the random sample correspond to each label (including additional superficial labels 'No label' and 'Multiple labels').

These are in fact the labels distribution over the random sample under the current hypotheses, so we can output them as a pie chart representing the **expected** labels distribution over the dataset:

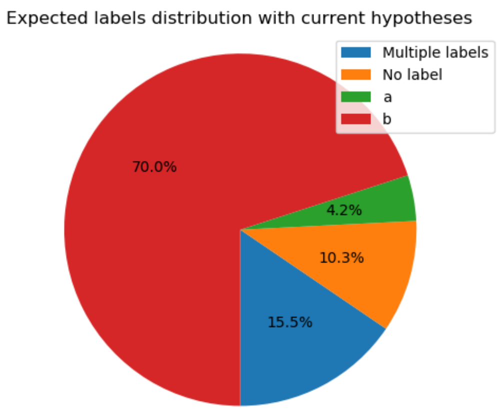


Figure 6.3: Expected labels distribution diagram, using only the random sample

Note that this is basically a visual representation of the (non-empty) clusters. Note also that we only show percentages in this case, since actual numbers (out of the size of sample) make no sense to the user. At this point they can look at the (new) hypotheses and consider whether they agree with what they have in mind when they do the manual classification. That, along with the stats displayed in the pie chart on the screen, will help them decide to use these hypotheses to classify the whole dataset or go back to the cycle in order to improve them. Obviously, with this pie chart we see more than a quarter of the models (in the sample) either have multiple labels or no labels at all, so normally a user would choose the latter. But let's say they choose the former and compare these results with the actual values we obtain.

6.2.2 Actual values

So as discussed above, the stats implementation at this stage is relatively straightforward: while classifying the dataset, count how many models have label 'a', how many 'No label', and so on. When the classification process is (finally) over, we ask the user whether they are happy with the outcome and wish to exit or whether they would like to go back to classifying more models. Previously, all they had at this point as means to asses how accurate the classification was were random choices of models in the dataset and manually checking if they were correctly labeled. However, using the label counters, we can help the user with the following pie chart:

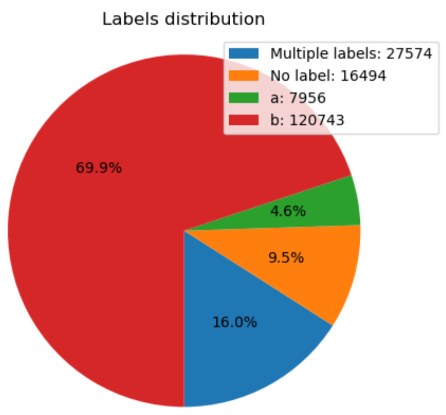


Figure 6.4: Actual labels distribution on the entire dataset

Two things to mention here:

- since this is no longer based on a small sample of the dataset, displaying actual numbers makes sense; the user might want to know the exact number of models labeled 'a'.
- we see that the expected values, based on a sample of 1000 out of 172.000+ models, are indeed quite accurate (70.0% vs. 69.9%, 16.0% vs. 15.5%). More on how the size of the sample affects these differences is discussed in the Evaluation chapter.

6.3 Pre-named components

Note that, as before, we will use component "name" and "type" interchangeably, as in our use case they are considered the same thing.

While implementing the detection and naming of components functionality, we discovered there was a slight issue with the generated models provided by the Siemens team: for every component, the model representation (in Prolog syntax) only showed ports that actually were endpoints of edges in the overall architecture graph. For example, the vehicle component does have a front_axle and a rear_axle port, but if a model happens to be front-wheel drive (so no edge connecting the rear_axle port), the rear_axle port is not showed in the model representation at all.

This would clearly not work with the component detection implemented so far, since the vehicle component is, in this case, structurally identical to, say an ICE: they both contain a single rotational port and nothing else. That means that if the tool detects this kind of component, the user might name it an "ICE", and all instances of vehicles with only one used port will be seen as an ICE, which is obviously not good.

Interestingly enough, with this clearly flawed implementation we still managed to get hypotheses with 99.995% accuracy on the 172.767 models (only 7 were mislabeled). However, this is clearly an accident with this particular dataset whose hypotheses thus seem to be somewhat oblivious to whether something is a vehicle or an ICE component. It shows that ILASP is also oblivious to these differences and can learn **surprisingly** accurate hypotheses given a problem and some background knowledge to use for learning, regardless of how correctly that background knowledge was written by the user.

Anyway, the clear solution to this problem, upon discussion with the Siemens team, was to allow for pre-named components in the Prolog definitions. Hence, we simply decided to extend the syntax for the supplied Prolog files, so that:

- $node(model\_id, node\_id)$ means component with id $node\_id$ is part of model with id $model\_id$.
- $node(model\_id, node\_id, comp\_type)$ means component with id $node\_id$ and type $comp\_type$ is part of model with id $model\_id$.
- no component is defined in a different way.

Now, the user has to *actively* let the tool know that the components are pre-named; they do that by running *classilasp* with the flag "-p":

```
$ classilasp -p path/to/architectures/file/architecturesFile.pl
```

Listing 6.1: Calling classilasp with pre-named flag

Obviously, in order to do this we need to slightly change the code of the *classilasp* script itself too:

```bash
#!/bin/bash
length=$(($#-1))
args=${@:1:$length}

pathArg="${!#}"
INPUTPATH=$(readlink -f $pathArg)

cd $(dirname "$0")
python ./modules/main.py $args "$INPUTPATH"
```

Listing 6.2: classilasp script with arguments

The only difference is that the dataset path is no longer the *first* argument to run our main module with, but rather the *last*, and everything else is considered a flag that the tool deals with internally.

One further thing to mention is that, in terms of error handling for pre-naming, there are two cases that the tool treats differently:

1. The '-p' flag is set, but there exists at least one component not pre-named. This breaks the assumption that **every** component is pre-named when this flag is set, the tool cannot know or deduce what the actual name is, so an error is raised and execution terminates. This is safe since this is still in pre-processing, no learning has taken place yet, no information gets lost, so the user is notified that they should either correct the dataset or run *classilasp* without the '-p' flag.

2. The '-p' flag is not set, but there exists at least one component that is pre-named. This is much safer than the previous case: since the flag was not set, the tool knows that it has to learn **all** component names from the user, and so it simply disregards any existing name arguments and proceeds as described in the previous chapters in order to identify and name components using user input.

6.4 Skip model

One thing suggested by the Siemens team was that, sometimes, the user may not be sure what the label for a model is. Without the possibility to skip that model, at least temporarily, they are "forced" to offer a label. If that happens to be wrong because of their uncertainty, the whole learning process will fail since the ideal hypotheses are no longer what we're converging towards.

From an implementation point of view, this is again not very complicated to do: have a list with "skippedModels" in the state module, and whenever the user decides to 'skip' (which is simply an extra option in the list of labels they normally have to choose for the model), add the model to the list and go back to choosing a model from the clusters.

We don't put them back in the clusters (yet) in order to make sure that they don't show up again. However, when new hypotheses are computed, they are indeed added back in the clusters (based on their new predicted labels); the reason for this is that these updated predicted labels may potentially help the user classify the model this time.

6.5 Optimizing parallel classifications

As mentioned at the beginning of the chapter, before we could implement something a little more complex, such as queries of the dataset, we absolutely needed to improve the whole classification process of the dataset, since most operations involving the entire dataset along with a set of constraints follow mostly the same pattern, and thus have the same complexity.

As previously discussed, running one model classification per *clingo* process ends up taking about 1 second for 10 models, which adds up to a total of five hours for the whole dataset of 170.000+ models. Also, it is unfeasible to instead put all models in one file and run that huge file in one single *clingo* process. However, this does touch on the idea of how to remove some, if not all, multi-processing unrelated overhead.

We could try to do a solution that's basically somewhere in between the two approaches from above: run a **maximum** number of models per process. Then we can make that number significantly larger than 1, thus hopefully improving the 5 hours running time, without making the number large enough to use too many resources.

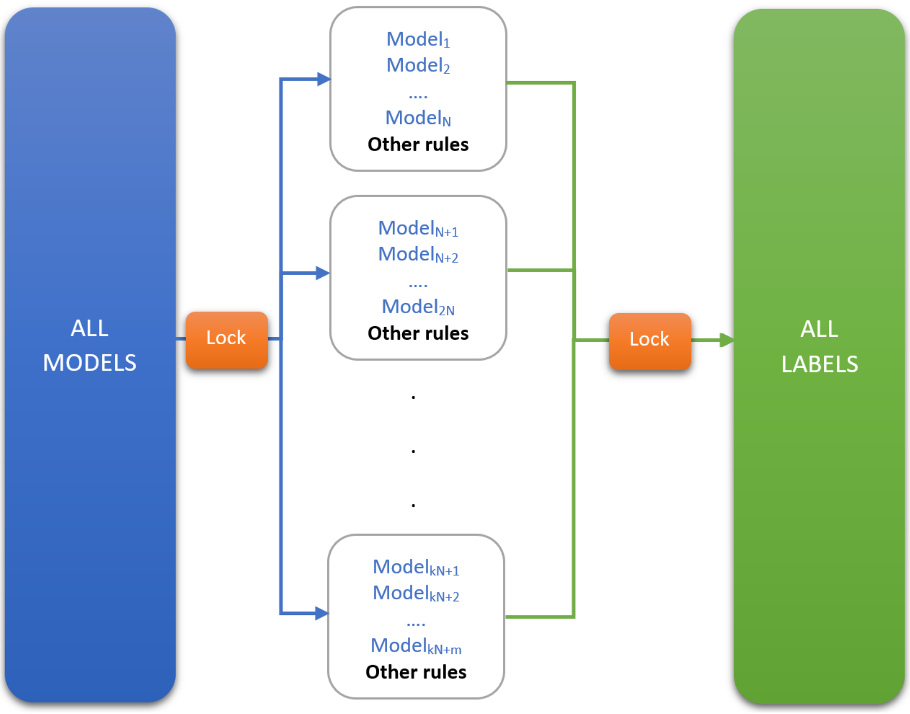


Figure 6.5: Classification: multiple processes with multiple models each

Here we see the general structure of this approach. On the left-hand side we see "ALL MODELS", which is the file containing the string representations of the given dataset. Then there is a fixed number of *threads*, in this case $k + 1$. Each one pops $min(N,\ size(allModels))$ models, where N is the (maximum) number of models per *clingo* process, and $size(allModels)$ is the number of remaining unclassified models. Note that all of this is done using a Lock, so that reading and popping from the dataset is thread-safe.

Once a thread has its models (or rather their string representations), it can release the Lock and process them. It then computes the string representing the *clingo* program involving these models, also adding **Other rules**. In the classification case, these would contain the background knowledge and the hypotheses. Once the labels are computed for all models through the *clingo* process, they are written to the output file, here depicted by the "ALL LABELS" box. Again, since these are write-to-file operations, we use a Lock to ensure thread-safety.

After this write operation is finalized, the thread is "free" to perform another series of classifications for some models and thus goes back to the input file. This cycle ends when no models are found in "ALL MODELS" anymore; in that case, the function ran by the thread returns. When all threads are done, the classification process is over.

6.5.1 Generalizing hypotheses

This cannot, however, work with the hypotheses as described so far. We've addressed this before, a hypothesis containing only rules of the form:

parallel :- body_of_rule.

might entail the atom *parallel* from N models in a *clingo* program, but from which model specifically? In other words, if we obtain the atom *parallel* in an answer set, we know for sure that at least one model is indeed parallel, but cannot know exactly which one. That's why we need to generalize these hypotheses in order to make them more "per-model", i.e. make them contain rules of the form:

label(M, parallel) :- body_of_rule.

where *M* represents a model id, and *body_of_rule* is now also dependent on *M*. There are a few ways to do this, but it's important to remember that we are adding another variable to potentially a lot of predicates, which will impact grounding and thus computation time. Since this is *clingo* classification of hundreds of thousands of models and not ILASP learning from a much smaller number of examples, every such small difference can have a big impact on the overall computation time.

This is why we again contacted the Siemens team and they assured us that, within every input Prolog file, all component (node) and port ids are different. Obviously, so far we only needed components to have different ids within one model, and since we only ran one model per *clingo* program anyway, that worked as expected. But if we have component id duplicates, a predicate involving only predicate variables would become confusing (e.g. $are\_connected(c1, c2)$ says $c1$ and $c2$ are connected; but if $c2$ is also the id of a component in another model, then it would "see" a connection between separate models, thus affecting other predicates, like $direct\_path$ and so on; in general, this is unreliable behaviour that we need to avoid).

Of course, globally different ids is something that can easily be enforced anyway, either by the user, programatically, when they generate the data or even by *classilasp* in pre-processing when parsing the input file. So making this assumption allowed us to make the optimization described above by only altering three kinds of predicates in the hypotheses:

- label atoms, e.g. "parallel" becomes predicate "label(M, parallel)"; note that we don't use simply M for the model predicate in the implementation, in case some rules already contain this variable, but for simplicity we shall use just M here.
- invented predicates. Note that these two items so far constitute all possible heads in hypotheses, which obviously have to also be per-model, and thus add M as an argument, e.g. "invented_pred" becomes "invented_pred(M)", both in the heads and the bodies of the rules it is part of.
- other predicates directly "tied" to the model id; so far, these are only the "comp" predicates, since they need to explicitly express that, if a hypothesis uses component with id "compId", the label(s) obtained from that hypothesis are attached to the model containing that component, say model with id "mId". In other words, we transform predicates "comp(compId, compType)" into "comp(mId, compId, compType)". The predicates directly "affected" by what the model id is are stored in a list in the constants file, though it's highly unlikely for that list to grow, since only components are this dependent on the model id.

It might not be immediately apparent how the hypotheses change using the three kinds of predicates from above, so let's consider an example. Take the following hypothesis:

```
% PARALLEL
invented_pred :- comp(V1, battery), direct_path(V0, V1, V2).
parallel :- not invented_pred.
parallel :- comp(V1, vehicle), direct_path(V0, V1, V2).
```

This is a hypothesis we obtained in the passive learning chapter, so let's see how this changes with what was described now:

```
% PARALLEL
invented_pred(M) :- comp(M, V1, battery), direct_path(V0, V1, V2).
label(M, parallel) :- not invented_pred(M).
label(M, parallel) :- comp(M, V1, vehicle), direct_path(V0, V1, V2).
```

For example, the last rule says that M is parallel if it contains a component ($V1$) of type $vehicle$ that is inside a $direct\_path$. However, the second rule says that M is parallel if $invented\_pred(M)$ does not hold. This seems unsafe: in general this does indeed **not** hold for most atoms in our program, for example anything that's not a model. So we obtain that something that's not even a model is parallel, which doesn't make sense. That is why we need to add, as a safety measure, predicate "model(M)" to the body of all hypothesis rules. This is safe to do since, as said before, all head predicates will also contain M, so we are simply grounding its value immediately to something sensible.

So finally, we obtain the following equivalent and now correct form of the hypothesis above:

```
% PARALLEL
invented_pred(M) :- model(M), comp(M, V1, battery), direct_path(V0, V1, V2).
label(M, parallel) :- model(M), not invented_pred(M).
label(M, parallel) :- model(M), comp(M, V1, vehicle), direct_path(V0, V1, V2).
```

Since all component ids are unique within the input file, $direct\_path$ does not need to contain M; if $direct\_path(c0,\ c1,\ c2)$ holds, $c0,\ c1$ and $c2$ must be part of the same model by the way $direct\_path$ is constructed. With hypotheses in this form we can now have more than one model in the *clingo* classification programs and still obtain each one's label(s).

Now, one might ask why we don't simply generate the hypotheses in this final form directly. The reason for that is that displaying hypotheses in ASP syntax is already asking a bit much from a normal user with no ILP background (it is, clearly, quite readable as it is, but there have been talks of trying to parse these hypotheses, process them and print their equivalent in English, but more on this later). So we made every effort to keep the hypotheses as short and succinct as possible, so that they're intuitive to understand by anyone. The final version of the hypothesis from above is, indeed, helpful for the tool's optimization algorithms, but it loses readability for the average user.

As already mentioned in previous chapters, this change yields a surprisingly large performance difference. If we now run 1000 models per *clingo* process, this means we run around $\frac{172000}{1000} = 172$ such processes in total, as opposed to the previous 172000, so about 99.9% of the multi-processing-related overhead should now disappear. As it turns out, that does make a big difference, as the overall computation time is indeed reduced from about 5 hours to 2 minutes.

This changes a few things:

- the user is no longer restricted to one, maybe two full classifications of the dataset per *classilasp* run, if they wish to finish the whole process in one day. They can now pretty much do a full classification whenever they wish to, which allows them to have a look at how the whole data "works" with the current hypotheses instead of relying on the expected distributions most of the time.
- any similar operation (like querying) can now be implemented and expected to take more or less the same amount of time, if performed on the entire dataset. However, they should also take into account the fact that the rules they use have to also be "per-model" (i.e. be converted to contain a model variable where appropriate).
- reclustering, which requires classification of around 1000 models in general, used to

take almost 2 minutes (1 second for 10 models); now it is instantaneous.

- one of the original goals we established in the first chapters - writing the tool in such a way that it doesn't add too much overhead to the actual learning process in ILASP - is now reinforced.

6.6 Queries

While working with the dataset of 170.000 models, we noticed that only 3 models were *series*; that's 0.001%. No matter how smart and complex clustering is, it is simply statistically impossible to get to one of those models "randomly" for the user to classify them (unless they're really, really lucky one time). Thus, with no *series* models, the *series* hypothesis will remain empty (since all examples will be "not *series*", which the empty hypothesis already classifies them as).

That's why the user bias **must** be used in order to obtain one such model: they know, at least loosely, what constraints a *series* model might satisfy and, say, a *parallel* one might not. For example, we noticed that *some* series models don't have *torque_coupler* and *planetary_gearset* components (though some others might). We also noticed that *some* series models tend to be relatively smaller, less complex than average. This is the kind of bias that we can use (and that the tool knows nothing about) in order to get to a series model and classify positively.

Once this became clear enough, the next question was: how do we obtain this bias from the user? In other words, what should be the "language" the user would use to express their bias for the tool to interpret and use? Again, we need to remember that the assumption is the user has no ILP background. However, we can at least expect them to be able to replicate rules as simple as the ones in the hypotheses, i.e. be able to write rules similar to the ones they *can* read and interpret in the hypotheses.

Another argument for this is that, if in the future hypotheses change their form to something else (e.g. translated to English), queries will simply change to whatever the "inverse" of the change the hypotheses suffer is (i.e. if hypotheses are translated to a "formalized" language through a clear, linear process, say P, then queries should be written in that language, and the corresponding rules should be inferred via P^{-1}). Of course, how this can be done is a whole different topic of its own with many possibilities, but currently beyond the scope of this discussion.

So for now we agreed that the queries should be written in a simple form, using the same predicates that appear in hypotheses (i.e. in the mode bias). Given that in hypotheses we have rules of the form:

series :- body_of_rule.
and
invented_pred :- body_of_rule.

queries should follow the same pattern. So we must allow for invented predicates, and have a keyword (equivalent of *series* above) that, if entailed by a model, means that model complies with the query rules; that's how classification normally works, if a model entails *series*, then the current hypotheses classify it as *series*. Since this notion of query is fairly similar to a database query, we used **select** as the keyword: if a model entails *select*, then we select it as part of the complying models.

So the theory is now complete, given the observations we mentioned above, we need to run the following queries which should, if we're right, lead us to a *series* model:

```
% QUERY #1
```

```
p :- comp(V, torque_coupler).
q :- comp(V, planetary_gearset).
select :- not p, not q.
```

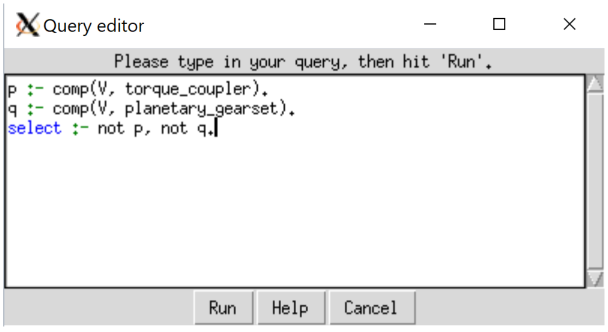
```
% QUERY #2
r :- comp(V0, _), comp(V1, _), comp(V2, _), comp(V3, _), comp(V4, _), comp(V5,
    _), V0 > V1, V1 > V2, V2 > V3, V3 > V4, V4 > V5.
select :- not r.
```

These are the ASP equivalents of the two observations about some *series* models:

- Query #1 says that (new) atom p holds if the model contains a *torque_coupler*, q holds if it contains a *planetary_gearset*, and **select** holds if neither p nor q hold, i.e. it doesn't contain either of those components.
- Query #2 says that r holds if the model contains at least 6 different components; **select** holds if r does not hold, i.e. if the model contains at most 5 components. Since there are models with 4 up to even 9 components, in order to choose the "smaller" ones, we choose those with 4 or 5 members.

Note how the translation into the English equivalents from above is basically the process P described before, whereas writing the actual queries from the observations (in English) was the clearly harder P^{-1} process.

It became clear that the best way for the user to input their queries would be to have a pop-up containing a text editor and a "Run" button, which closes the pop-up and runs the query. Doing this in the terminal would otherwise be not only more complicated to implement, but also a worse experience for the average user. In Python, the best and most common tool to do that is TkInter [11]. With a **Text**, a **Scrollbar** and three **Button** widgets, we obtain the following:

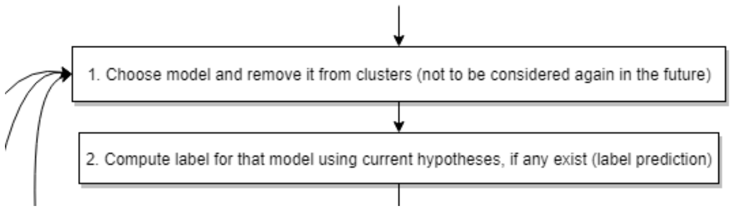


```
p :- comp(V, torque_coupler).
q :- comp(V, planetary_gearset).
select :- not p, not q.
```

Figure 6.6: Basic query editor

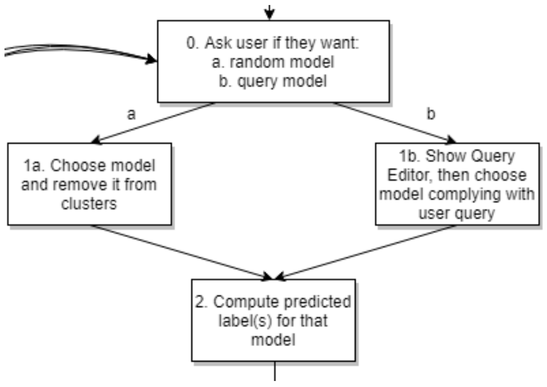
The Text widget also allows us to do some nice things that are pretty much the standard for any text editor, such as highlighting certain words or patterns (as seen above with **select** or "**:-**"), as well as having certain shortcuts such as *Select All* (Ctrl-A), *Copy* (Ctrl-C), *Paste* (Ctrl-V), and so on.

But when exactly should we show the Query Editor? Clearly, the user must manifest their will to no longer receive a "random" model, and instead get one based on their query. That means Step 1 in the flowchart before is now no longer simply "Choose model and remove it from clusters", but instead it requires an extra question for the user: do they want a model from the clusters (so a "random" one) or do they want one based on their custom constraints? If they choose the former, they proceed as before; if they choose the latter, they are shown the editor, they input their query, and then they instead get a model complying with it (that may or may not be in the clusters). In other words, the initial steps of the flow chart change as presented in the following figures:



1. Choose model and remove it from clusters (not to be considered again in the future)
2. Compute label for that model using current hypotheses, if any exist (label prediction)

Figure 6.7: First cycle steps without querying



0. Ask user if they want:
a. random model
b. query model
a
b
1a. Choose model and remove it from clusters
1b. Show Query Editor, then choose model complying with user query
2. Compute predicted label(s) for that model

Figure 6.8: First cycle steps with querying

Note how the entry point is now Step 0, when the user is asked to choose either a random model or a query one, and every step that previously "sent" execution back to Step 1 now goes to Step 0, since the user input is always required at this stage.

Anyway, as previously said, when the "Run" button in the Query Editor is clicked, we take the string representing the query and use it in an almost identical way to how classification worked:

1. Generalize the query to contain a model variable where appropriate. In this case, as with classification, only the head predicates (i.e. **select**, plus all newly declared predicates by the user) and the *comp/2* predicates qualify, so the process can be done generically, in the exact same way as before.

2. Run "querying" on multiple threads, each one popping at most a fixed number of models each time (e.g. 1000), adding their string representations along with the (generalized) query to a file, then run it with *clingo*; add all models for which $select(M)$ holds to a "complying" list.

3. When all threads have finished (i.e. when they all "notice" there are no more models to use and return), the "complying" list contains all models the user wants.

One thing to mention is that there is obviously no need to add to the "complying" list any model already labeled by the user. That is why we keep (in the state) a list of the ids of the models already labeled, so that we can enforce this observation. Since the user won't label a large number of models and the model ids are very short strings, this list will barely take up any resources during execution.

6.6.1 Error handling

Now, before we can investigate the results of a query, we should make sure the overall execution of the tool is safe, no matter what the user writes in the Query Editor. This means we have to deal with all kinds of errors that might arise.

- The first kind of "problem" we need to take into account is not necessarily an error, but it is nevertheless a special case: pressing "Cancel" or the "X" button. The latter simply kills the whole editor window, and thus no matter what the Text widget contained, we get an empty string for the query (since the "Run" button was never pressed). This is actually quite a stable and intuitive behaviour, so we can enforce the same to happen when the user presses "Cancel" and also in the separate case when they click "Run" with an empty string for the query.

  Consequently, when we get the query string back from the Query Editor in the tool, we can have a special case for an "empty" query. Then, the query is obviously not run at all, and instead the user is alerted and then sent back to Step 0, as described above.

- For actual errors, it is very convenient that we run the query with *clingo*, which obviously comes with all the error handling we need. As a result, the tool is in this case a mere intermediate between the user and *clingo*: it takes the string program from the user and "supplies" it to *clingo*, and in turn, if any error arises, it can simply show it to the user. Whenever this happens, the user receives the error and is sent back to Step 0.

As can be seen, the tool **must** make sure that no matter what error arises in the Query Editor, it will not crash the whole app, and is instead handled locally and appropriately.

6.6.2 Query cache

Now let's remember why we used a random sample instead of the whole dataset throughout the learning cycle: since the user is not expected to classify a very large number of models, it is safe to select only a sample that's large enough to be sufficient, but still relatively small overall. Also, it needs to be completely random, so that it can be seen as representative for the dataset. We now again run into the same problem: the list of models complying with a query can potentially be very large (in general, worst case is the entire dataset!). So by the same reasoning, we should proceed similarly; storing the entire list in the memory is inefficient and simply not necessary.

That is why we can keep a subset of this list and get rid of the rest. As long as this subset is a random sample, it is also representative of the entire list. Let's call this subset the *Query cache*, or simply the **cache**, and allow it a fixed maximum size, say 100 (in the tool this is a custom value stored in the constants module).

One question that arises is: why do we need to go through the entire dataset, find ALL complying models and only then select 100 at random, when we can simply "short-circuit" the search once we've found 100? The reason is that, in general, the models in the dataset are expected to not be in a truly random order; given their nature, they were instead most likely combinatorially generated. Given how this type of algorithms work, it can be expected that generation starts with "smaller" models and then incrementally builds up to the "large" ones. This results in a file with relatively smaller models in the first half than the second half, for example. So if we short-circuit after the cache is full, it might be that we only selected smaller models and didn't get to find some more complex ones, which might be exactly the ones the user is interested in. That is why we go through the whole file (now that it no longer takes hours to do so), and only then take a representative sample to place in the cache.

Another obvious question is: why call it "cache" when it's no more than a random sample of a list? That's because that is, in fact, one of its main attributes: if the user doesn't change the query and directly clicks the "Run" button, then there's no need to rerun the query, since we know that it's going to give us the same result. So in that case, we directly pop a model from the cache and return it immediately.

One small, but important observation is that the user can still skip a model coming from the cache; however, that model will not go in the "skippedModels" list in the state, it will simply be disregarded. The reason is that, every time new hypotheses are computed, this list is emptied and all its contents are put back in the new clusters, in case the user can now classify them using the new predicted labels. However, if we added "query" models to this list, then they would end up in the clusters. We don't want this; the clusters only contain models from the random sample, which were selected with no bias whatsoever, therefore completely at random. This is not true for the "query" models, since they are by definition chosen using user bias, and therefore adding them to the clusters, they would become part of the sample, making the sample no longer "random", and therefore not representative for the whole dataset. Thus, it can no longer be used to compute *expected* values reliably.

6.6.3 Saving queries

Let's have another look at one of the two queries presented before:

```
r :- comp(V0, _), comp(V1, _), comp(V2, _), comp(V3, _), comp(V4, _), comp(V5,
   _), V0 > V1, V1 > V2, V2 > V3, V3 > V4, V4 > V5.
select :- not r.
```

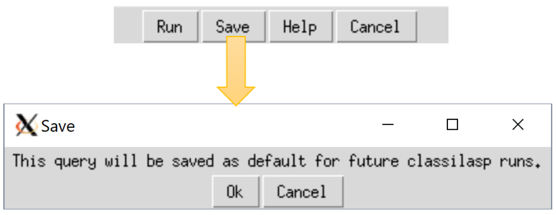
Clearly this is quite long and not exactly trivial. Once the user types it in and clicks "Run", the cache gets populated and the user is presented with a complying model. If that's not one of the models they want (e.g. they get a *parallel* model which also happens to have at most 5 components, but they wish to get to a *series* one), they will likely choose the skip option, and go back to the Query Editor. It is quite annoying that they now have to type the entire query again and again, until they eventually get to the model they want. This brings us to the two types of *query saving* features presented below.

Temporarily

This is the immediate solution to this problem: whenever the user types in a query, clicks "Run" and the query passes all checks for errors, the query is saved in the state as the "previousQuery" and then run in the normal way. That way, next time the user opens the Query Editor, that query will appear by default in the text editor and they can modify it or "Run" it as it is to get another model from the cache.

Permanently

If we look again at the query from above, we notice that r is, in fact, a very general and useful predicate; it simply says that the model has at least 6 distinct components. It doesn't use any component names or anything specific to the current dataset, so this makes it an important query predicate candidate for the future. However, even though it is *temporarily* saved after the query is run, it gets lost once the tool execution terminates, so it will need to be rewritten for any other *classilasp* runs in the future.



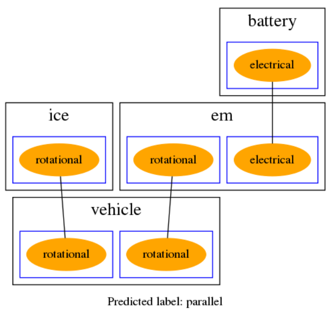
This query will be saved as default for future classilasp runs.

Figure 6.9: Save pop-up

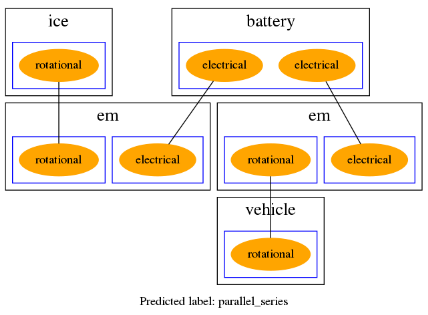
That is why we added the option to save queries permanently. As can be seen in the figure, when the user clicks the new "Save" button, they are asked if they're sure they wish to save the current query as the default. Note that this only makes the current query the "start-up default", i.e. the query that appears the very first time the user asks for the Query Editor in a *classilasp* run. From then on, *temporary saves* take precedence, so that the user can repeatedly use any predicates they define specifically for their current dataset.

Because this is entirely *clingo* syntax, the user can take advantage of that and use comments. That is not because the queries might get complex enough to actually need comments (one would assume), but rather to comment out some predicates if the user so chooses. That way, the permanently saved query can contain several important predicates and the user simply chooses which ones to use and which not using comments. Because the default query is saved in a file of its own, called $defaultQuery.las$, these generic predicates can be defined by one person in such a file and then everyone else can import that file to be used as their own default.

Before we move on to the next section, we present below two models complying with the two specific queries presented in this section, designed to lead us to a *series* model:



(a) Parallel query result model



(b) Series query result model

Figure 6.10: Query results

Note how both of them comply with the two queries, but only the second is indeed a *series* model. If we initially get the one on the left-hand side, we can simply skip it, as it is a *parallel* and would likely not help improve the class hypotheses (since even its predicted label is correct, as observed in the diagram). After skipping it, we can again go to the Query Editor, and because we don't change the previous query, now *temporarily* saved there, we simply get another model from the cache. Eventually, we get to the model on the right-hand side, which is the *series* model, incorrectly predicted as *parallel_series*.

Finally, it should be noted that we get to the *series* model after skipping much fewer models with Query #1 than with Query #2, which obviously proves that, even though both queries successfully lead us to a model we want, one is more effective than the other (by having fewer complying models).

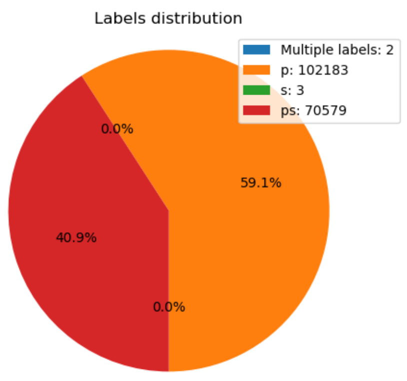
6.7 Prioritizing certain models

This is also based on a couple of suggestions from the Siemens team, once again in order to further speed up the learning process.

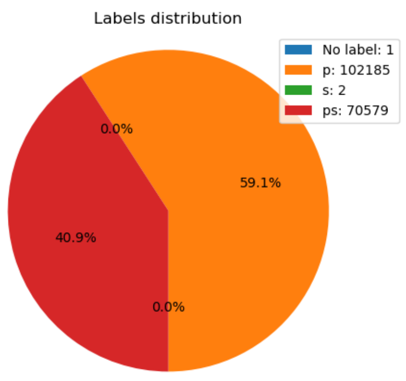
6.7.1 Based on predicted labels

Although we've already discussed a certain prioritization of models with no or multiple labels within the clusters, the team proposed that we add the option for a user to ask specifically for one such model. This is quite trivial: when a special cluster is non-empty, add a corresponding extra option to Step 0 from above; e.g. when the 'No label' cluster is non-empty, add question *"Generate model with no predicted label for classification?"*. If the answer is yes, pop model from the 'No label' cluster in the normal way.

However, there is another important type of prioritization that we can do here. Take, once again, the classification process of the 170.000 models, and let's say we've arrived at a set of promising hypotheses that prompt us to use them in order to classify the whole dataset. Once that is done, we get one of the following distributions:



(a) Only 2 'Multiple labels' models in the whole dataset



(b) Only one 'No label' model in the whole dataset

Figure 6.11: 'Almost' perfect label distributions

Upon inspection, in both cases we notice that the only models that are mislabeled are the ones with multiple/no labels. However, because there are only at most 2 such models out of 172.767, the chances of them being in the random sample of only 1000 models are, once again, basically 0. This also means that, at the moment, the current hypotheses label **all** models in the sample correctly. So if we're to find the *ideal* hypotheses, we must somehow keep track of those few models in the overall dataset that are guaranteed to be mislabeled.

We do that by, as usual, adding 'No label' and 'Multiple labels' models that we find when doing full classification in two lists (respectively) that we keep in the state. Because these lists can become very large in general, we also restrict their size to a maximum value, stored in the constants file.

Now, it is fairly obvious that the 'No label' models found in the full dataset are more "important" than the ones we currently have in the clusters, for the simple reason that the former are most likely not in the clusters. Therefore, they cannot be generated "randomly" during the learning cycle; the only way the user could get to them would be through a query, but they have no idea what those models are, all they know is that they exist and have no predicted labels, so the user cannot come up with the query to find them.

Hence, because these models are more important than the ones in the clusters, whenever the user asks *specifically* for a 'No label' model, we choose either from the 'No label' cluster or from the 'No label' full-dataset-list, but with a larger weight for the latter (same principle goes for 'Multiple labels' models). That way we can easily find the outlying models in the distributions from above and eventually converge to the *ideal* hypotheses.

6.7.2 Based on size

In general, "smaller", less complex architectures are expected to be easier to classify. That is because, whatever rule/pattern/substructure the user must look for should be easier to spot in such models. However, it should be noted that some such patterns can sometimes only be present in the more complex architectures and therefore cannot be learned from the smaller ones. Therefore, we should prioritize the smaller models, but not choose exclusively from them.

We do this in a very simple, but effective way: whenever we must choose a model from a given list (cluster, cache, etc.), we sort that list based on number of components - which is done instantaneously because these lists are always of relatively small size, as we've always made sure - and divide the list in three equal parts. Clearly, the first part contains the relatively small models in that list, the second contains the medium-sized ones and, finally, the third contains the most complex ones.

We now randomly choose the model from the three parts, but with different weights: first part has weight 3, second part - 2 and third part - 1. These are adjustable and may change in the future based on user experience. Currently, they ensure about half ($\frac{3}{3+2+1}$) of the time we get small models, one in three times ($\frac{2}{3+2+1}$) we get a medium-sized, and the remaining ones are relatively more complex.

6.8 More complex patterns and substructures

So far we've used the terms "patterns" and "substructures" without defining exactly what we refer to. Now, with everything that's been implemented, we're in the position to let ILASP learn more complex patterns and substructures, so it's best to define them and then look at some such examples.

So let's remember how $direct\_path$ was defined (informally):

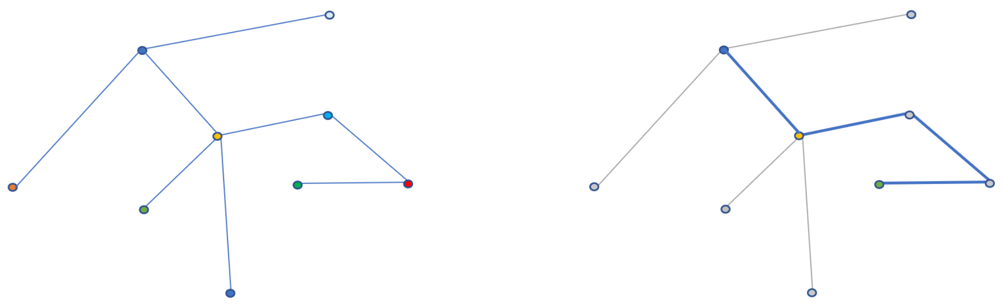

(a) Model graph (b) Emphasized $direct\_path$

Figure 6.12: Model graph with emphasized $direct\_path$

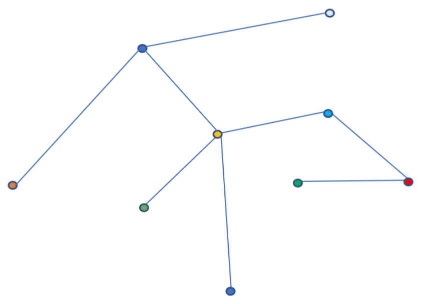
Given the model graph above, we can see that there is a $direct\_path$ from the "dark blue" component to the "green" one, through a "yellow" component, i.e. the following holds:

$$comp(c0, blue).$$
$$comp(c1, yellow).$$
$$comp(c2, green).$$
$$direct\_path(c0, c1, c2).$$

So although the *blue* and *yellow* components happen to be connected, $direct\_path$, as its name suggests, describes a "path", so a **pattern** that can be as long as possible within the graph. It can be more localized too; let $c3$ be the other $green$ component (the one connected to the $yellow$ component). We notice that the following also hold:

$$comp(c3, green).$$
$$direct\_path(c0, c1, c3).$$

With this in mind, we define a **substructure** to be more localized than a pattern. Take the following example with the same graph from above:


(a) Model graph

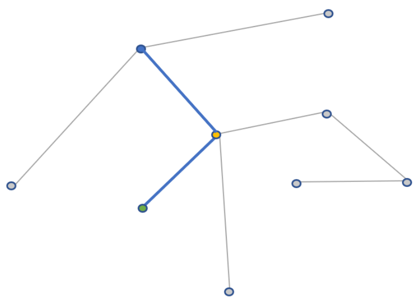

(b) Emphasized *direct_line*

Figure 6.13: Model graph with emphasized *direct_line*

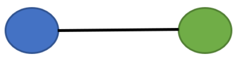
So we see that the "equivalent" *substructure* form of the *pattern* $direct\_path$, called here $direct\_line$, is in fact very similar to it. The only difference is that it is completely local, meaning that it only involves **direct connections** via edges, rather than **(arbitrarily long) paths**. This means that the following still holds:

$$direct\_line(c0, c1, c3).$$

However, the following no longer does, since $c1$ and $c2$ are not connected, as defined above:

$$direct\_line(c0, c1, c2).$$

As it now becomes a bit clearer what the difference between a *pattern* and a *substructure* is, it should also be noted that any substructure qualifies as its "equivalent" pattern, but in general the reverse does not hold (since patterns involve arbitrarily long inner paths instead of direct connections via edges). To make the distinction, in the following graphs we shall again draw edges as straight lines and arbitrarily long basic (non-cyclic) paths with a squiggly line:


(a) Direct connection via (single) edge

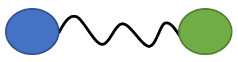
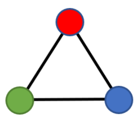

(b) Arbitrarily long basic path

Figure 6.14: Direct connection vs basic path notation

With this distinction, we can now present some more complex substructures/patterns that ILASP can learn. For example, one can define the notion of a **triangle**. As a substructure, that simply represents three components that are inter-connected, clearly local to some part of the architecture they are part of. As a pattern, this means there is a basic cycle (with no other "inner" cycles formed by its nodes) containing those three components. To emphasize again, clearly the triangle *substructure* follows this *pattern*, but in general not the other way around. So the two versions can be represented in the following high-level forms:


(a) Triangle substructure

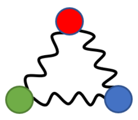

(b) Triangle pattern

Figure 6.15: Triangle substructure vs pattern

The triangle substructure is fairly straightforward to define:

```
triangle(C1, C2, C3) :- compV(C1), compV(C2), compV(C3), C1 != C2, C2 != C3, C3
    != C1, are_connected(C1, C2), are_connected(C2, C3), are_connected(C3, C1).
```

It simply says that it involves three distinct components that are inter-connected. The definition of the corresponding pattern is a little more complicated though:

```
triangle_path(C1, C2, C3) :- compV(C1), compV(C2), compV(C3), C1 != C2, C2 != C
    3, C3 != C1, basic_path(C1, C2, M), basic_path(C2, C3, N), basic_path(C3, C
    1, P), val(C1, A), val(C2, B), val(C3, C), M & N == B, N & P == C, P & M ==
     A.
```

This follows the same ideas from the (formal) definition of $direct\_path$. Let's consider the following visual interpretation of the code above:

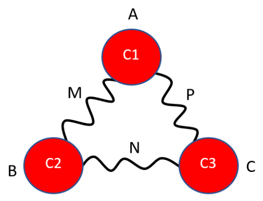
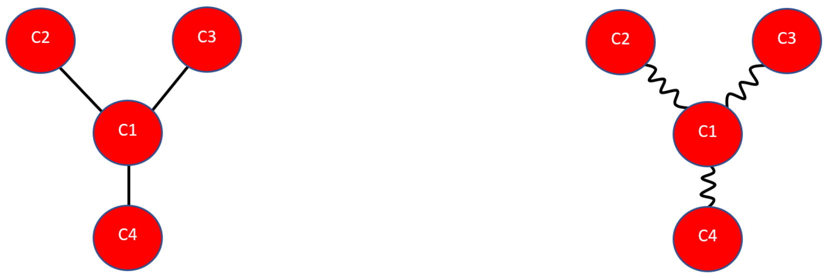


Figure 6.16: Triangle pattern definition diagram

So we have three distinct components, $C1$, $C2$ and $C3$, with binary values A, B and C respectively. We also know that there is a $basic\_path$ between $C1$ and $C2$ with code M, another between $C2$ and $C3$ with code N and finally one between $C3$ and $C1$ with code P. Similar to the reasoning from the $direct\_path$ definition, $M\&N == B$ implies that the previous basic paths from $C1$ to $C2$ and $C2$ to $C3$ intersect **only** in $C2$. The remaining two checks yield similar results, which finally tell us that the basic paths from before, encoded with M, N and P respectively, only intersect in $C1$, $C2$ and $C3$. Given their definitions, this means that $C1$, $C2$ and $C3$ do indeed form a basic cycle, as described above.

Obviously, this cycle is no longer local in an architecture, and can instead potentially even span the entire graph, reinforcing the idea of a pattern.

That being said, substructures and patterns can obviously get more and more complicated when defined in a very similar fashion. In fact, the Siemens team told us that one heuristic they used when automatically classifying their data involved a "Y" shape. Whether they meant a local substructure or a more general pattern, it's still interesting to define both in case either is needed for future classifications. This will now involve 4 nodes, and will look like this:



(a) "Y" substructure (b) "Y" pattern

Figure 6.17: "Y" substructure vs pattern

Before we move on, let's consider one last question: does the graph we've seen a few times before (and again right below) contain a "Y" pattern with an *orange*, a *blue* and a *green* component as "extremities", regardless of the component in its "middle"?

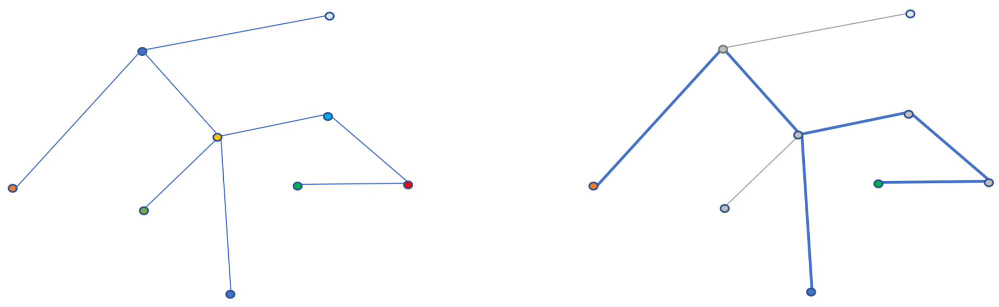

(a) Model graph (b) Emphasized "Y" pattern

Figure 6.18: Model graph with emphasized relevant "Y" pattern

As it turns out, it does, but we can see how the more complicated the pattern becomes, the more difficult to spot it is for the user (it is even more difficult in general, since, as we've seen already, graph nodes are not simple nodes coloured differently, but quite large components with inner ports that make the actual connections, edges may "intersect" each other in the diagrams we generate, so the patterns truly become difficult to visually identify for a person). As both graphs and patterns/substructures start involving more and more nodes, manual classification may end up taking minutes per model instead of seconds, as mentioned in the Introduction.

Now, there is one question about how substructures are defined that needs to be addressed: since all substructures are almost entirely defined in terms of the predicate $are\_connected$, can we not have that predicate alone in the mode bias without having to define additional substructures in the background knowledge? That way, the substructures, if present in the models, can be learned as well. Technically, yes, ILASP can learn all substructure predicates using $are\_connected$ (it would also need to have an additional predicate to express that components are distinct, which also adds to the hypothesis space's complexity).

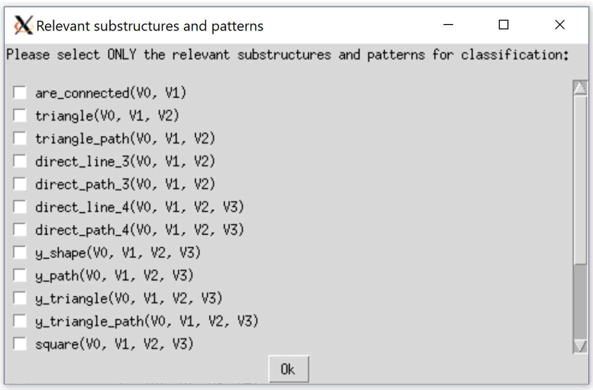
However, as we're about to see, instead of adding the predicate $are\_connected$ alone to the mode bias and specify a sufficiently large number of instances that it can be used in a rule, we can use the user's bias to intentionally restrict the search space to only those substructures that are relevant. After all, the purpose of ILASP in *classilasp* is not to learn substructure predicates that we can simply supply directly in the background knowledge; that would simply add to the computation time without learning something truly relevant for the class hypotheses, i.e. how these substructures are actually used to define them.

6.8.1 Visual mode bias for the user

When adding new substructures and patterns, we obviously need to tell ILASP to look for them. We do that by adding them to the mode bias. However, if we add all of them, ILASP will likely end up taking hours, even days, in order to compute new hypotheses. The reason is that, while the mode bias grows linearly with every new added substructure/pattern, the hypothesis space it (combinatorially) generates will end up growing more or less exponentially.

However, in general only a very small subset of substructures/patterns may be needed for a given task. Once again, the only one who knows that is the user, so we need to use their knowledge to decrease the size of the search space as much as possible. We clearly cannot ask them to write the mode bias; having them read and write simple hypothesis/query rules in ASP is already asking a lot from a user with no ILP background. Requiring them to write the bias too is simply too much, so we need to do this in a more user-friendly manner.

One thing to notice is that all predicates that can go in the bias are either $comp$, or substructures/patterns which only contain *component* variables. So we can simply ask the user which substructures/patterns they think are relevant for the given task, then we can deduce their full form to be added to the bias. Specifically, in pre-processing we use another simple TkInter pop-up, displayed below:


Relevant substructures and patterns

Please select ONLY the relevant substructures and patterns for classification:

- [ ] are_connected(V0, V1)
- [ ] triangle(V0, V1, V2)
- [ ] triangle_path(V0, V1, V2)
- [ ] direct_line_3(V0, V1, V2)
- [ ] direct_path_3(V0, V1, V2)
- [ ] direct_line_4(V0, V1, V2, V3)
- [ ] direct_path_4(V0, V1, V2, V3)
- [ ] y_shape(V0, V1, V2, V3)
- [ ] y_path(V0, V1, V2, V3)
- [ ] y_triangle(V0, V1, V2, V3)
- [ ] y_triangle_path(V0, V1, V2, V3)
- [ ] square(V0, V1, V2, V3)

Figure 6.19: Relevant substructures/patterns pop-up

Obviously, diagrams of all existing substructures and patterns must be part of a manual supplied to the user, so that they can know exactly what each predicate represents. For now, they can be found in Appendix A.

The user simply checks the relevant boxes (however many they wish, potentially all of them), the selected items are stored in a list in the state, and every time an ILASP or *clingo* program is created, only those predicates are used. The easiest way to store their corresponding information is to have a directory for all substructures/patterns, and a background and bias file for each. The background file simply contains the definition of the predicate. For example, for the triangle pattern, called *triangle_path*, the background file will contain the definition previously presented, while its bias file contains:

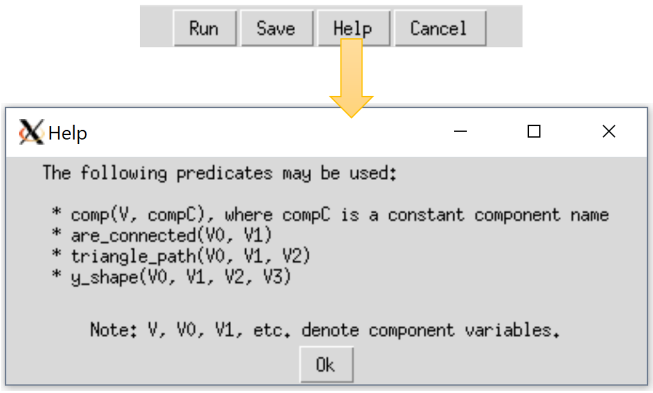
```
1 #modeb(1, triangle_path(var(compV), var(compV), var(compV)), (positive)).
2 #bias(":- body(triangle_path(V, _, V)).").
3 #bias(":- body(triangle_path(V, V, _)).").
4 #bias(":- body(triangle_path(_, V, V)).").
```

Lines 2, 3 and 4 are a way to further restrict the kinds of rules that are generated in the hypothesis space. In this case, they don't allow any generated rule to contain in their body the predicate *triangle_path* with two equal variables, since by definition that is not possible, so there's no need to generate such a rule.

6.8.2 Query customized help

When covering queries, there was a small thing that we intentionally left out, because it is only now that it makes sense to explain it: the "Help" button in the Query Editor. Normally, the user might forget what predicates they can use in order to write their queries; that's where this feature comes in. Initially, the list of these predicates was very small, basically only containing *comp*, *are_connected* and *direct_path*. Now, however, the user has a much larger list of substructures/patterns they can use, as well as full control over *which* of those are actually relevant.

Since after pre-processing, the tool keeps track of this list, it can present it in the "Help" pop-up. If, for example, the user thinks that only *y_shape* (the "Y" substructure) and *triangle_path* (the triangle pattern) are relevant, then this is what they'd see if they clicked on the "Help" button:



```
The following predicates may be used:

  * comp(V, compC), where compC is a constant component name
  * are_connected(V0, V1)
  * triangle_path(V0, V1, V2)
  * y_shape(V0, V1, V2, V3)


       Note: V, V0, V1, etc. denote component variables.
```

Figure 6.20: Help pop-up

6.9 Noise

Adding the noise functionality to *classilasp* is in fact quite easy: we need to run ILASP version 3 instead of the usual 2i, since it performs better with noise, as well as "noisify" the example files, i.e. add penalties for each example. The outputted hypotheses are in the exact same form, so they can be displayed and used in the exact same manner. There are, however, two ways to use noise in *classilasp*:

6.9.1 Running *classilasp* with noise

In order to do this, we ask the user to use a special flag: '-n'. The tool then updates the 'noise' parameter in the state to **True**. That way, it will know exactly which version of ILASP to use, as well as how the examples should be constructed (with penalties or not). Penalty values are again customizable, being stored in the constants file. For the moment, the penalty is always equal to 100; the value is chosen to be large enough not to be affected by the size of the hypotheses, i.e. no matter how big hypotheses get (likely not to get to 100 predicates), hypotheses covering the **largest** number of examples will always be preferred to the smaller hypotheses covering less examples. Since the size of hypotheses is pretty much always expected to be less than any individual example penalty, this is clearly enforced.

6.9.2 Switching to noise in UNSATISFIABLE case

This was actually the behaviour the Siemens team required to be the default, which is why we ask the user to *actively* manifest their will to have noise from the beginning by using the '-n' flag. Before implementing this, whenever we got "UNSATISFIABLE" from one of the ILASP tasks, we simply exited execution. The reason was that clearly no viable hypotheses could be found from that point on, so there was no reason to continue the search process.

However, we can now catch any such instance of "UNSATISFIABLE". Whenever this happens, we ask the user if they wish to exit (potentially to try to find an *exact* set of hypotheses with different relevant substructures/patterns) or if they want to switch the *noise* flag to **True** and find the set of hypotheses that covers *the most* examples they've provided so far. If they choose the former, simply exit. If they choose the latter, we have to "noisify" the example files that currently contain non-noisy examples (without penalties). Once we do this, we can rerun the tasks, this time with noise, and get an actual set of hypotheses instead of "UNSATISFIABLE", and then keep running ILASP with noise until the end of execution.

Chapter 7

Evaluation

Throughout the previous chapters, we've tried to provide the theoretical and mathematical reasons behind some of the decision we've made in the implementation. Now it's time to have a look at actual evaluations of how good those decisions were and perhaps see where we could've done better.

7.1 Quantitative evaluations

Here is where we discuss mostly statistics in terms of performance, how different values of certain variables in *classilasp* affect its computation time, reliability and even correctness. As before, we shall use the dataset with 170.000 models, as we're now familiar with the notions there, and its considerable size is also ideal for our following statistical experiments.

7.1.1 Sample size

The sample size clearly affects not only performance of most operations, but also how reliable the values we obtain are. Clearly, the sample size should ideally be some orders of magnitude smaller than the size of the full dataset (at least when that dataset is very large). Whether in general we should compute the sample size as a fixed *number* or a *fixed* proportion of the size of the dataset is an interesting first question.

Clearly, the first one is safer: if the sample size grows with the dataset, we might end up with a very large sample, which defeats its purpose in the first place. However, if the dataset size is very large, say tens of millions, choosing a fixed small number, say 1000, may not be as statistically representative as a *proportion* and hence give us unreliable expected values.

In any case, such large datasets are, in general, the exception to the rule, and since the size of the sample is customizable from the constants file, the user can change it to something they consider sensible if this really becomes a problem. So for this section, we'll have a look at how *classilasp* performs with different **fixed** sizes of the sample. Clearly, we only need to consider the cases where the sample size is indeed relevant; so far, only pre-processing (i.e. the MeanShift algorithm) and reclustering (i.e. computation of predicted labels for the sample) qualify. Other operations, such as popping a model from the clusters, are done in $O(1)$, so are independent of the sample size, even though they use the sample directly. Finally, for each sample size we can check how "accurately" they predict the labels distribution, as an expected value, through the pie charts we normally generate.

Sample size 10

Clearly, this is only for the sake of the argument, a sample of 10 models out of 170.000+ is not only in no way representative, but is not even sufficient for classification, since the user will likely run out of models very quickly.

As expected, MeanShift is done very quickly, in just 0.035 seconds, and one reclustering operation takes 0.144. However, when we look at the discrepancy between the expected and actual labels distribution, we yet again see why such a small sample is simply not feasible:

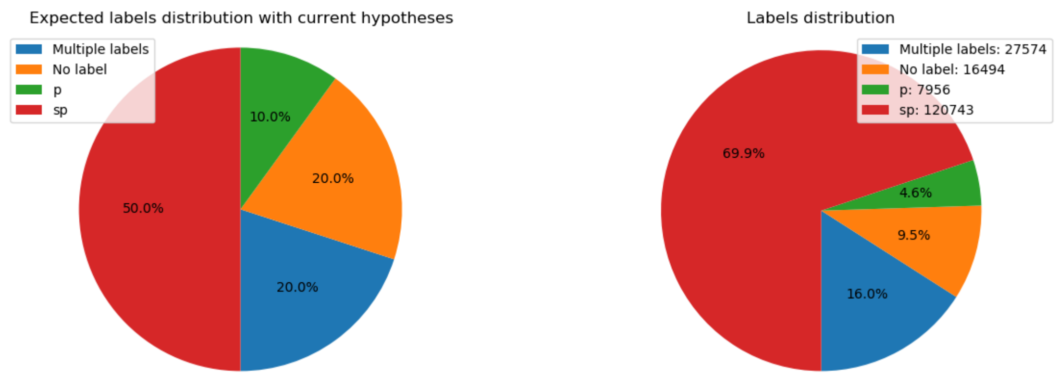


Figure 7.1: Expected vs actual labels distribution with sample size 10

The difference for the *sp* class is of almost 20%; that's 20% of the total amount of models, meaning an inaccuracy of around 35.000 models out of the 170.000, so clearly this is no way can be seen as a reliable expected value.

Sample size 100

This is still quite small, and in some cases the user may still run out of models; they shouldn't ever have to classify 100 models in order to get decent hypotheses, but given that they can also skip models along the way, it becomes a real possibility. Also, 100 models constitute only about 0.05% of the whole dataset, so that might still be too little to be considered reliable. MeanShift now takes 0.098 seconds, so it's still instantaneous to the user, while reclustering takes about 0.228. If we now consider expected versus actual distributions:

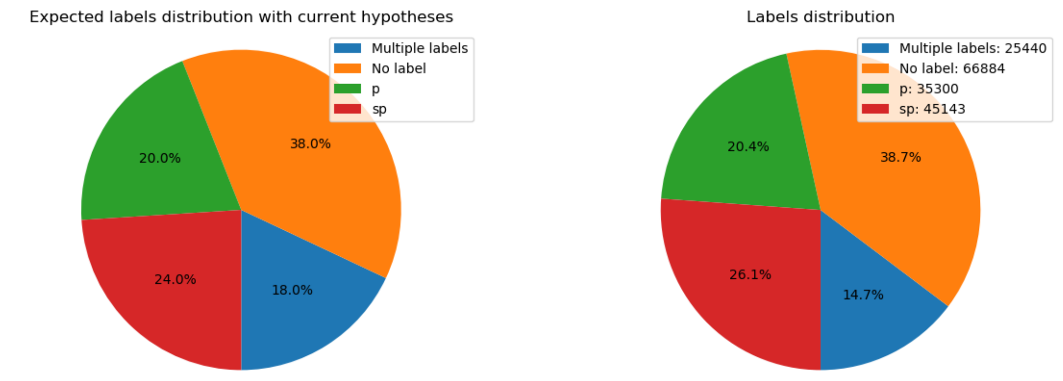


Figure 7.2: Expected vs actual labels distribution with sample size 100

We see how a sample size of only 100 yields surprisingly decent results, the biggest inaccuracy being of only 3.3% for the 'Multiple labels' portion. Out of the full dataset, that's about 5700 models. Still an amount we can't really consider "negligible", but for a sample of only 0.05% of the dataset, this clearly proves that **random** sampling can indeed be very effective.

Sample size 1000

This has been the default value we've used throughout the rest of this report, so now we can see it in action against the other "competitors". In terms of computational time, it takes 0.94 seconds for the MeanShift algorithm in pre-processing, and about 2.23 seconds for reclustering. Clearly, we're now getting into "full" seconds per these computations; this is ok for now, but for the larger orders of magnitude it's clear that computation time will be problematic. In terms of accuracy of the expected labels distribution, there's also an improvement:

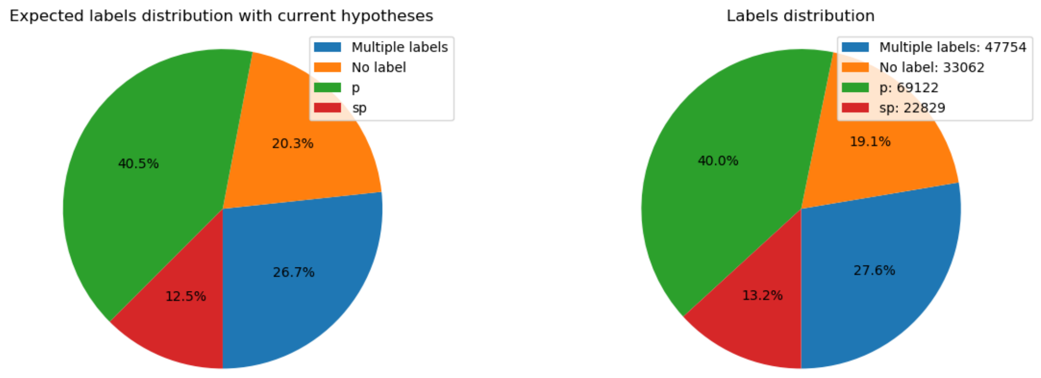


Figure 7.3: Expected vs actual labels distribution with sample size 1000

Even visually, the two graphs are almost identical; that's because most "differences" are between 0% and 1%, which account for at most around 1700 models. Given that the user is interested in proportions rather than numbers (since they won't know the exact numbers for the class sizes), we can see already why a sample size of 1000 has been the one we've used so far: it simply yields the best result with still very little computation time.

Sample size 10000

Finally, let's consider one last order of magnitude for the sample size and see how it performs: 10000. Note that this is almost 6% of the entire dataset. With it, MeanShift now takes much longer: 74.52 seconds, while an average reclustering operation takes about 10.79 seconds. This simply takes unnecessarily long; given the accuracy a sample size of 1000 displayed, there's no good reason to go for such a large value instead. This is also backed by the fact that this accuracy is not noticeably improved in the first place:

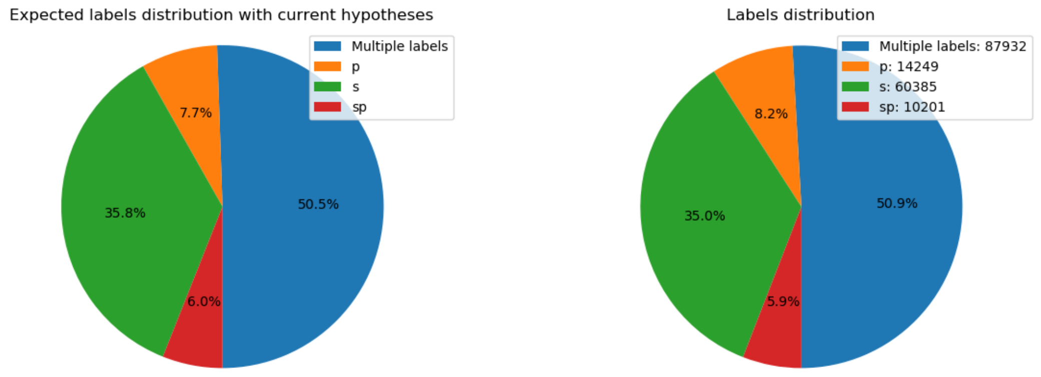


Figure 7.4: Expected vs actual labels distribution with sample size 10000

In conclusion, we can compute the standard deviation of the label distributions for these cases and analyze how it improves in contrast with the MeanShift computation time with each different magnitude level (i.e. base-10 log of the sample size):

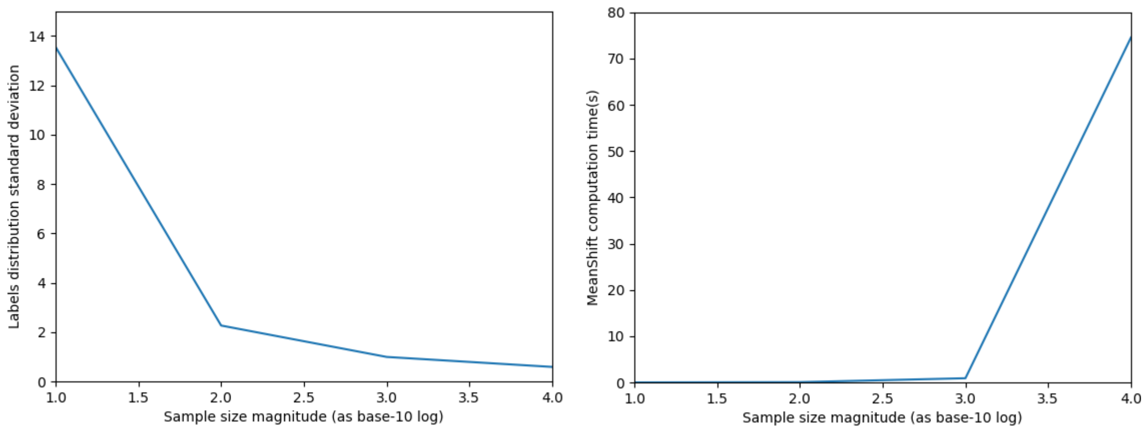


Figure 7.5: Standard deviation of label distributions vs MeanShift computation time (s)

Hence, the sample size is indeed configurable in the constants file, but based on the observations here, the current value of 1000 seems to offer the best results for the lowest computational costs.

7.1.2 Parallelization

We've seen how most tasks that *can* be run in parallel are now indeed run so. For example, reclustering, analyzed above, is not only dependent on the sample size, but because it is run in parallel, it depends on the values parallelization is configured with.

We've already seen how big the impact of too few or too many parallel threads/processes, as well as how many "tasks" we do per process, (e.g. how many models we classify at once with *clingo*) can be on performance.

So let's now analyze how changing some of these values actually affects performance. Before we can do that, we need to remember the operations we currently do in parallel:

- Since all label hypotheses are more or less independent from each other, we run all ILASP programs separately and in parallel, each in one process.
- Since the classification of one model using a set of hypotheses is completely independent from the classification of another, we run these in parallel as well. However, we don't just run one model's classification at a time (so one per process), but instead multiple models in each process, while still independently in principle. Note that we include reclustering in this part, since it also requires classification of models (done in parallel), but of a subset of the dataset, namely the sample.
- Queries are also done in parallel, and as emphasized in their section, they can be considered equivalent to how classification works, even though they serve a different purpose. Therefore, their analyses will also coincide.

Hence, let's analyze each item more closely:

1. Firstly, we have one ILASP program per label, and here there is indeed absolutely no reason to not run each in a separate process, given the relatively small number of labels we have (knowing that the user has to manually supply each of these labels, we can make the assumption that they're not going to supply thousands, for example).

We've also previously explained why it's best not to create one big ILASP program that could compute 2, 3 or even all hypotheses at once; that would not compute the hypotheses in parallel, so we would lose that advantage, but the even worse aspect is how large the hypothesis space would have to be in order to cover multiple labels. So there's not much statistical analysis we can do with regards to the parallelization of the ILASP label programs; keeping them completely separated is the best option.

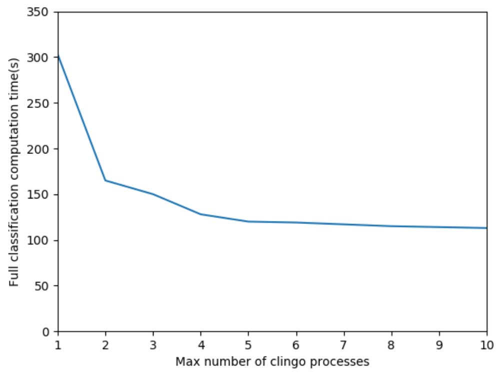
2. On the other hand, running classifications and queries in parallel is much more interesting and worth analyzing. For one, these depend on two variables:

   - **maximum number of processes**: that is, the maximum number of *clingo* processes we allow to be run at the same time; currently this is set at 5 in the constants file.
   - **maximum number of models per process**: this was discussed in section 6.5 and refers to the maximum number of models we allow to be classified in each *clingo* process; this is currently set at 1000.

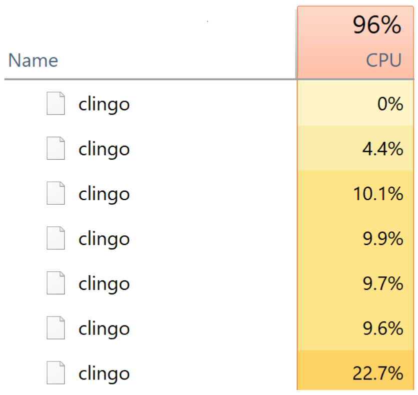
   The best approach to reasoning about these two variables is to fix one and see how computation time depends on the other. Clearly, all other variables discussed in this section take their default values unless stated otherwise.

Changing max number of processes

So in this case we fix the maximum number of models per process to the current default - 1000. Clearly, we cannot run a lot of processes at a time (not in the real sense) anyway, so let's see how computation time varies with values from 1 to 10:



(a) Seconds per full classification, as function of max number of *clingo* processes



Name	96% CPU
clingo	0%
clingo	4.4%
clingo	10.1%
clingo	9.9%
clingo	9.7%
clingo	9.6%
clingo	22.7%

(b) Resources used at peak of **7** concurrent *clingo* processes

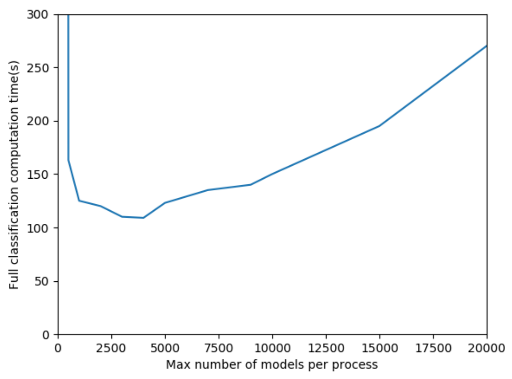
Figure 7.6: Computation time vs used resources #1

So on the left-hand side, we see how parallelization does help with reducing the *average* computation time for a full classification of 170.000 models (so also for queries and other similar features). However, we can also see how this time actually stabilizes to around 2 minutes, as we obtained in the previous chapters, and this happens quite early on, when using only 4-5 maximum processes, even though it continues to slightly decrease thereafter.

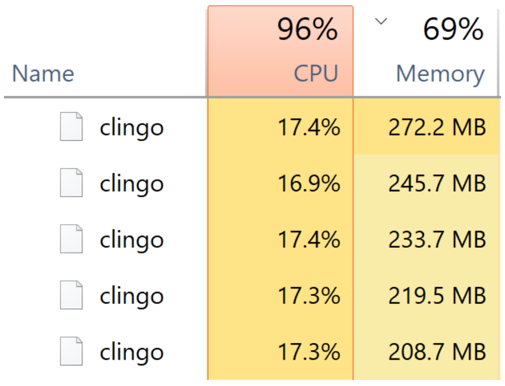
That is a good enough reason not to go for a much larger max size, but we get another good reason from the right-hand side figure: whenever these processes run truly in parallel, they can be quite a burden on the CPU, which the user may or may not accept (that's why this value is customizable). In the case above, 7 *clingo* processes take up almost 70% of CPU usage, bringing it to almost full overall usage. This also explains why the computation time stabilizes: it's simply unlikely that a large number of *clingo* processes will be allowed to ever run at the same time anyway.

Changing max number of models per process

In this case we now fix the max number of *clingo* processes to the default value of 5 and see how computation time varies with the max number of models we allow each process to use:



(a) Seconds per full classification, as function of max number of models per *clingo* process



Name	96% CPU	69% Memory
clingo	17.4%	272.2 MB
clingo	16.9%	245.7 MB
clingo	17.4%	233.7 MB
clingo	17.3%	219.5 MB
clingo	17.3%	208.7 MB

(b) Resources used by all 5 concurrent *clingo* processes, with max **7000** models per process

Figure 7.7: Computation time vs used resources #2

Here we see quite an interesting graph on the left-hand side, displaying a rather parabolic shape. Going towards 0, this makes sense: we know from before the optimization discussed in 6.5 that with one model per process this took around 5 hours, so around 18000 seconds. We also see how computation time decreases up until we reach a value of 2500-3000, then starts increasing again.

The reason for this becomes quite clear with the right-hand side figure: the more models we place in a single process, the more complex each computation gets; we are computing the answer sets of thousands of models at once, each with a fair number of predicates. So this requires more and more CPU and Memory resources, which means we no longer depend solely on the efficiency of our algorithm, but also on more low-level details, such us the operating system we are running this on and how it allocates these resources to each process, as well as even the underlying hardware specs.

So the reason why we decided on the value 1000 is fairly simple: it keeps each process relatively "small" in terms of the resources it requires, while still being very close to the optimum computation time (it takes 125 seconds on average, while the optimum is about 110 seconds).

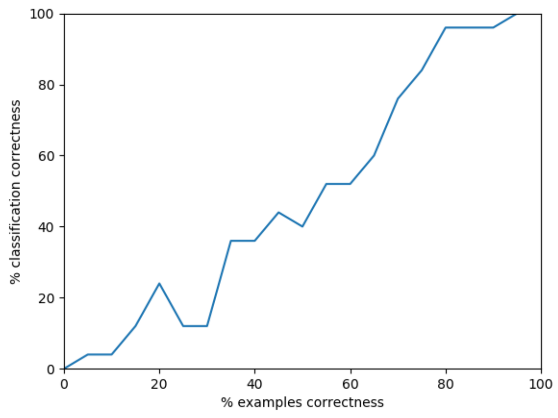
Of course, as emphasized already, these results do depend on the user's machine and different users will see different values (but similar shapes) for the graphs above; if they think computation takes too long without using much of their resources, they can naturally increase the maximum number of *clingo* processes and/or the number of models per each process.

7.1.3 Noise

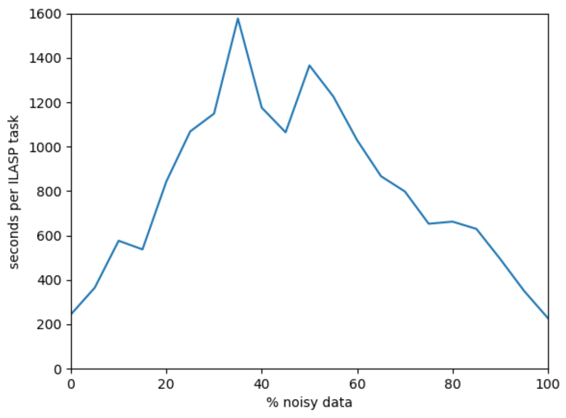
Basically, we've seen how noise works in theory, and because of the large penalties we've assigned per example, we do expect the **best** hypothesis in terms of examples coverage to be outputted by ILASP. However, in this section we want to see exactly how much noise is *too much*? In other words, how many mistakes (as a proportion) can the user make and still remain fairly certain they will get the *right* set of hypotheses? Note that neural networks are, in general, quite reliable from this point of view; we've chosen to use ILASP for a lot of the advantages it offers, but we need to prove it is also reliable in this sense.

So note that in this section we're not really evaluating *classilasp* per se, it's more of a relative evaluation with respect to ILASP, since *classilasp*'s role is to simply set the "noise" flag in the state, create the ILASP programs and then let ILASP actually figure out how to deal with the noisy data.

In any case, the method we use is as follows: we select 5 random samples from the large dataset, each containing 5 models (we shall call each sample a "chunk" for simplicity). For a given level of noisy data (ranging from 0% to 100%), we learn a hypothesis using 4 chunks and then test its accuracy on the fifth. We do this for each of the five chunks and compute the average accuracy. That will be the computed accuracy for the chosen level of noise. Repeating this process for each possible level of noise will give us the analysis data we require:



(a) Classification accuracy as function of correctness of examples



(b) ILASP computation time as function of proportion of noise

Figure 7.8: Noise performance analysis

On the left-hand side graph we analyze how reliable classification is in terms of the amount of noisy data. We see that for $95 - 100\%$ correct examples (so at most 5% noise) we got perfectly accurate hypotheses for classification. From then on, classification accuracy decreases as the noise level increases, as is expected, in what seems to be a linear manner, but with quite large deviations along the way. This is likely because of the size of the samples we used. We had to restrict the size of each chunk because each ILASP program uses 4 chunks in order to learn hypotheses for the classification of the fifth; each chunk having 5 models means each ILASP program uses 20 examples.

As can be seen, computation time is indeed also dependent on the level of noise, having a parabolic shape with its maximum around the middle, where one task can take as much as half an hour; given the large number of programs we had to run for this "experiment", this adds up quite rapidly and thus makes larger samples unfeasible.

In any case, these graphs tell us plenty already, especially through their symmetric shapes. Firstly, note that a level of 100% of noise means that we are virtually learning the negation of the hypothesis for 0% noise. For example, the hypothesis we learned from the (relatively small) chunks for *parallel* with 0% noise was:

```
invented_pred :- comp(V0, battery), direct_path_3(V1, V0, V2).
parallel :- not invented_pred.
```

The hypothesis ILASP computed for the same chunks with 100% noise was its exact negation:

```
parallel :- comp(V0, battery), direct_path_3(V1, V0, V2).
```

To generalize this aspect, it is expected that the left-hand side graph is symmetrical with respect to the point $(50\%,\ 50\%)$, as it is the case now, with a certain level of deviation depending on the size of the chunks.

It's also an interesting fact that whenever the user makes a mistake and thus generates noisy data, they not only decrease the accuracy of their hypotheses with respect to classification, but they effectively also increase the computation time of that best hypothesis, as the right-hand side graph clearly shows (unless the user makes more mistakes than not, in which case they will eventually start decreasing computation time back).

7.2 Qualitative evaluations

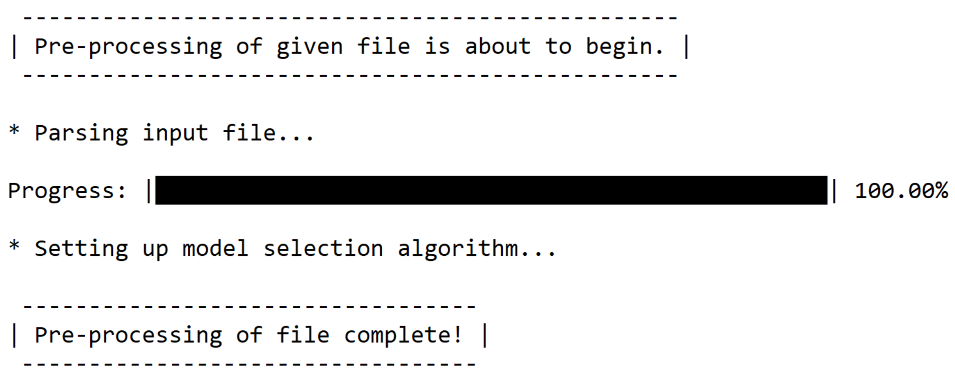
In this section we discuss more subtle aspects of the project that are not necessarily assessable through statistics and diagrams, but are still very important and relevant.

7.2.1 User interactions

In terms of user interactions, we wanted to keep them to a minimum while still obtaining as much relevant information from them as possible. The main actions the user has to do are: providing the **labels**, i.e. the classes for which ILASP has to learn the hypotheses, providing the **patterns** and **substructures** they think are relevant, and then obviously they have to provide their manually computed labels for a set of architectures the tool chooses in the active learning cycle. All other interactions are basic and used to dictate the flow of the application as a whole, as was presented in the flow chart of *classilasp*.

We haven't included the query functionality in the list above, where the user clearly has to provide a little more information. Note that the user can use this feature if they so choose, but they don't have to, in general, in order to do a complete classification with *classilasp*. We've explained why we decided to use ASP basic syntax as the "language" of communication between the user and the tool here. In general, this works fine and as with most Logic-based languages, it is quite intuitive to write the queries, even without ILP experience. This still felt like a compromise though, since one of the goals we set out at the beginning was to keep all interactions completely user-friendly for any user (with some assumed engineering background), and asking them to read ASP hypotheses and write ASP queries partially breaks this "promise".

Regardless, it's important to remember that *classilasp* involves **inter**actions, so the user must also know what the tool is doing whenever their input is not required. That is why we try to always provide that to the user, so that if an operation takes relatively longer to finish, the task's purpose, as well as its continuous progress, is shown to the user. So, for example, whenever we parse the whole dataset or fully classify it or perform a query, we display something like:



```
 -------------------------------------------------
| Pre-processing of given file is about to begin. |
 -------------------------------------------------

* Parsing input file...

Progress: |██████████████████████████████████████████████████| 100.00%

* Setting up model selection algorithm...

 ----------------------------------
| Pre-processing of file complete! |
 ----------------------------------
```

Figure 7.9: Pre-processing information

We also tried to make the flow of the application as intuitive as possible. That meant we had to avoid any specialized terminology wherever possible and instead simply direct the user generically through the cycle. Given the overall simplicity of the concept of active learning, it's not exactly difficult for a user to follow what each task's purpose is.

7.2.2 Testability

When the project became less "research-oriented" and more "implementation-oriented", this aspect clearly became a concern. Knowing that a module for tests would be needed at some point, this simply was another good reason for choosing to have a separate module with the application state and make everything else stateless.

As a result, most of the functions were either pure functions - i.e. they take a set of parameters of inputs and use them and them **only** in order to produce an output without altering anything outside of their scope - or "almost" pure, in that the only thing they alter outside of their scope is the state, so we can simply also check how the state changed in every test. It's also always very clear which functions are of which type, so that makes all these functions unit-testable. For the unit tests, we use the **unittest** Python framework, and here is a sample of how to do that:

```python
    def testCheckListsHaveSameElements(self):
        l1 = [2, 3, 1, 2]
        l2 = [1, 2, 2, 3]
        self.assertEqual(utils.checkListsHaveSameElements(l1, l2), True)

    def testCheckListsDontHaveSameElements1(self):
        l1 = [2, 3, 1, 3]
        l2 = [1, 2, 2, 3]
        self.assertEqual(utils.checkListsHaveSameElements(l1, l2), False)

    def testCheckListsDontHaveSameElements2(self):
        l1 = [2, 3, 1]
        l2 = [1, 2, 2, 3]
        self.assertEqual(utils.checkListsHaveSameElements(l1, l2), False)
```

Listing 7.1: Unit tests

However, there are obviously some much more complicated functions, e.g. involving multi-threading or user input (so with potentially non-deterministic behaviour). Those are unfortunately not unit-tested, at least for now, and their behaviour's reliability is only tested mostly by hand.

7.2.3 Extensibility

We were lucky enough not to have any point when we got truly stuck while implementing new features and optimizations, and what helped the most was the decision to modularize the project's structure as much as possible; whenever an item or function seemed to be independent in functionality and implementation, it was separated.

We also tried to never settle for the first, quickest and potentially very specialized solution for certain problems, and instead always took the time to think of the more general aspects behind them; for example, when developing the classification feature, we weren't yet aware it might resemble the functionality of some other features that would be developed in the future, such as reclustering or queries. However, we tried to write this feature in the most generic way possible, so that when we came to actually implement queries, we were able to reuse a lot of the functions previously written for classification.

Another positive aspect was, again, having the state module, but this time for a slightly different reason. Having no modules with states of their own meant fewer dependencies we had to worry about: everything only ever depended on the **single state** and nothing else.

That way, whenever a new feature had to be integrated, there was always just this one place we had to look to for the needed information, be it the current hypotheses, the list of labels to update, the last query, and so on. Hence, the state works as a very intuitive "communication" hub for all other modules, which makes integration of new features much easier and safer.

Chapter 8

Conclusion

In this chapter we reiterate the main points discussed throughout the previous chapters, then present some of the more interesting and useful extensions for *classilasp*.

All in all, all optimizations and separate features aside, this project can really be seen as writing a "wrapper" cycle around ILASP in order to use its learning capabilities in a dynamic, interactive process with the user in order to classify a set of typed graphs.

Since we've essentially accomplished this task, at least for the datasets we've worked with, we can now state some of the things we've learned as a result:

- **Answer sets** are indeed a powerful concept and can be used in ILP, with a powerful tool such as **ILASP**, in order to classify very large datasets with very little training data, being able to even account for **noisy data**.

- **Clustering**, along with the *incremental reclustering* technique that we've used, as well as other **prioritization** "heuristics", can greatly speed up active learning and are therefore highly recommended for such tasks.

- Given that the whole classification process is based on the user's knowledge, full use of it should be made, especially by allowing them to **query** the dataset for models they think are important/relevant as opposed to only providing them with random models. Their bias is also valuable when deciding on the **mode bias**, i.e. when choosing which patterns and substructures are relevant for classification and which aren't, in order to improve the chances of finding correct hypotheses, as well as the computation time to do so.

- Though its use can make a massive difference in computation time, it's not sufficient to *just* use **parallelization** whenever it seems sensible to do so, and instead a lot of investigative work and analysis should be done before deciding on the exact details of its implementation; it can make the difference between 5 hours and 2 minutes.

- **Highly modularized** applications are very easy to extend, so this should always be the preferred philosophy of development. Where appropriate, the application **state** should also be isolated as much as possible in order to improve extensibility and testability.

- It is important to tackle the **most general representation** of a given problem, so that the solution is applicable in potentially other similar fields. For example, a surprising type of classification *classilasp* should be able to handle with its now vast variation of patterns is the following:

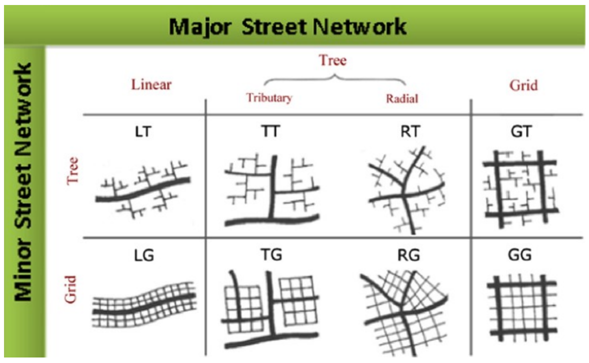


Figure 8.1: Street patterns classification

Clearly, we can define these street **patterns** using our binary encodings and $basic\_path$ predicate, considering the intersections as nodes. In fact, we might not even need to define them fully, since there are even more "high-level" bold patterns that differentiate some of those patterns from each other (for example, as the diagram suggests, we can easily notice the higher-level patterns of the **Linear** structures as opposed to the **Tree** or **Grid** ones). However, some specialized definitions are indeed clearly needed in order to differentiate between patterns of the same "Major Street Network" classes, for example GT versus GG. These definitions can either be explicitly written in the background knowledge or not, in which case ILASP would have to deduce them. For example, with a simple "star" pattern and a "square" pattern, the entire RG pattern can be defined (admittedly, with a very large number of variables though).

Note also how these are untyped graphs. Because our algorithm works with typed graphs, we can simply assign the same type to all nodes; this, in fact, greatly improves the computation time in ILASP, since there is only one *constant* component value, and the number of such constants makes the hypothesis space grow more or less exponentially with them.

However, there are ideas and suggestions that couldn't be implemented within the time frame established for this project; they are listed below as possible future extensions.

8.1 Future work and extensions

Of course, new emerging techniques in the ILP field may also add some further interesting extensions to *classilasp*, but for the moment these are the most important ones:

8.1.1 Functionality extensions

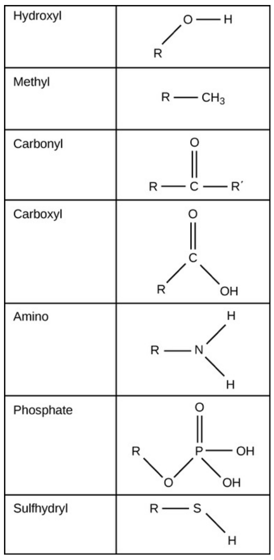
- When running *classilasp* with noise, and hence ILASP too, we may want to investigate whether **custom example penalties** are appropriate. The reason is based on the experience we've had with our datasets, specifically (once again) the 170.000, where tens of thousands were *parallel* models, tens of thousands were *series-parallel*, but only 3 were *series*. This indicates that the user mislabeling a series architecture is much more important than them mislabeling one of the other classes; the problem is that the tool cannot know that the user mislabeled a model, and more specifically it cannot know what the real, correct label was. So if an *actual* series model gets labeled as parallel, the tool cannot know that a **very** important model has just been mislabeled and thus treats it as any other model.

  If, however, an *actual* non-series model gets labeled as series, what should the tool do? Assign it a larger penalty because it's a positive example, of which it knows there are very few? That wouldn't work well, since that would have just assigned a large penalty to a mislabeled model, thus making it harder to exclude. Assign it a smaller penalty then? But if it was a correctly labeled series model instead, then because there's so few of them, it would risk being "ignored" and thus not covered by the hypotheses and specifically not have a correct *series* hypothesis.

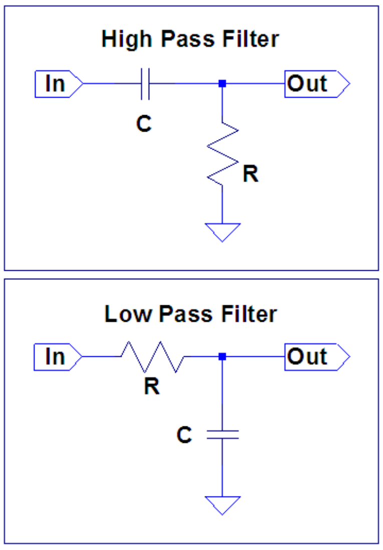
It is, hence, an interesting aspect to think about, though we haven't yet arrived at a consensus, which is why, for now, we stuck with equal penalties for all examples.

- So far, it became clear that in order to be as "aggressive" as possible in our strategy to speed up the learning process, many kinds of prioritization had to be used. However, they were always based on some heuristic, like model size or predicted labels (with the current hypotheses). These **prioritization rules** may actually be learned using one of ILASP's best features, that we've presented in the Background chapter, but haven't yet used: **weak constraints**. Whether done in a separate helper tool or in *classilasp*'s pre-processing stage if the user sets a given flag, a learning process of these weak constraints may be done and the output saved in a file for future use. That way, we can prioritize models not based on fixed heuristics, but on customized rules learned with the user's bias.

- As said before, one goal for *classilasp* is to be applicable in multiple, unrelated fields. Though we've seen one such field (other than automobile architectures) where it could work, here are a couple of examples where the patterns it has cannot be used and would need to be generalized slightly:



(a) Classifying functional groups of organic molecules [2]



(b) Classifying electronic RC filters [1]

Figure 8.2: Graph pattern classifications in other fields

Note how there are some patterns in the left-hand side table that *classilasp* has already covered. For example, the first item, namely Hydroxyl, is defined simply using the **direct_path** predicate:

```
hydroxyl :- comp(V1, o), comp(V2, h), direct_path(V0, V1, V2).
```

The reason why *classilasp* can't yet deal with most of the other patterns in the two figures is: it currently accounts for undirected graphs where there can be at most one edge between any two components. Some of the left-hand side "graphs" need *classilasp* to have patterns containing multiple edges between components, whereas the right-hand side needs those patterns to account for directed edges. Once that is done, the relative simplicity of these patterns should still make them easy enough to recognize.

8.1.2 Usability extensions

- It's clearly harder to interpret user queries in something other than ASP syntax, at least for the moment, but we should at least try to have a parser for the hypotheses in order to interpret them (since they have a clear, finite and well-known set of predicates they can use) and output a basic equivalent form in something **more human-readable** by the average user, such as actual English.

- One of the initial objectives was to develop the tool as an **online** application; however, as the project evolved, there never seemed to be a really good reason to do that. The tool was instead implemented to work offline, on the user's machine. Whether it will make sense in the future to put it on a server instead is something still worth considering, though this wasn't very relevant to the main topic and scope of this project, so it became less and less of a desired feature.

- A lot of the user interactions require a "back-and-forth" between the Terminal and certain pop-ups and diagrams images that appear in separate windows. This is not exactly desirable from a user experience point of view, so it might be a good idea to fully move all of *classilasp* in a **centralized GUI**.

- Remember how, when getting UNSATISFIABLE from one of the ILASP programs, the user is asked if they wish to exit or continue with noisy data assumed. It may be worth to also give them the option to **change the relevant substructures/patterns** and continue classification with the new ones, but still without noise. The reason this would be a good idea is that if they exit execution, all classified examples will be lost (since they are stored in the temp directory), so if they rerun *classilasp* with the same dataset, but different relevant substructures and/or patterns, they will virtually start the learning process from scratch, when they could have used the already classified examples.

- At the moment, *classilasp* only has one pre-processor, which expects Prolog files. Most users may store their datasets in more natural formats, such as JSON or xml files, so it might be worth writing **separate pre-processors** for those formats as well and let the user choose. Given the modularized architecture of *classilasp*, "connecting" a different pre-processor should be easy enough, as well as developing it with a lot of reusable methods from the older pre-processors.

User Guide

Installation

Before installing *classilasp*, please make sure you install ILASP and *clingo* **exactly** as advised at [27], including the recommendations. Note that, for the moment, *classilasp* is only implemented for Linux distributions, and is, thus, using the according ILASP version (it was, however, developed using the Linux Bash Shell on Windows, so it will also work with that). Please also make sure to have the latest Anaconda version, in order to have all the available Python libraries that are used in *classilasp*.

Once that is done, you can download *classilasp* from its GitHub repository at:

https://github.com/cristianchirac/classilasp

How to run *classilasp*

classilasp is run using the script with the same name, found in the root directory of the repo above. You can run that script with its path or you can add that path to your $\$PATH$ variable or to $/usr/local/bin$, as with *clingo* and ILASP. If running on Windows, please download, install and run Xming [12] beforehand, as well as the following command:

```
$ export DISPLAY=:0
```

Listing 8.1: Setting DISPLAY

Once you've done this, then you can run *classilasp* with your architectures file as follows:

```
$ classilasp [options] path/to/architectures/file/architecturesFile.pl
```

Listing 8.2: Calling classilasp

For specific details on how the input file should be formatted, we strongly advise you to consult section 5.2.1. As of writing this report, there are only two options *classilasp* may expect:

- **-n**: this signals *classilasp* that noisy data may be expected, so that ILASP is run with noise as well (please consult [27], as well as sections 3.3.5 and 6.9 for more details on this)
- **-p**: this signals *classilasp* that **ALL** the *node* predicates have a third argument, describing their type. If not set, the user may be prompted to name each different component *classilasp* finds. Please consult section 6.3 for more details.

Sample *classilasp* run

We will now do a sample *classilasp* walk-through in order to see how basic interactions work:

1. We begin by running *classilasp* with our input file, as described above. For this example, we run it with the '-p' flag and without the '-n' flag, meaning the input file contains only

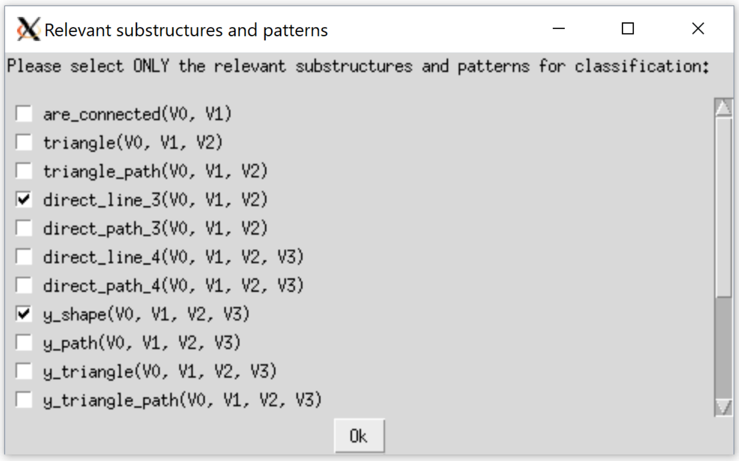
pre-named components and that all our manual classifications should be regarded as not noisy, respectively. If no errors occur in pre-processing, we should get the following:

```
 -------------------------------------------------
| Pre-processing of given file is about to begin. |
 -------------------------------------------------

* Parsing input file...

Progress: |██████████████████████████████████████████████████| 100.00%

* Setting up model selection algorithm...

 ---------------------------------
| Pre-processing of file complete! |
 ---------------------------------
```

Figure 8.3: Complete pre-processing output

Note that, without the '-p' flag set, this step would additionally require us to name any component diagrams that are shown on the screen. This is would not be done for every single component in the input file, but only for every *different kind* of component (in terms of what ports they contain, their port groups, etc.)

2. Next, we have to tell *classilasp* what the relevant patterns/substructures for classification are (described in Appendix A). Please select **only** the ones relevant, since computation time is highly dependent on their number:



Relevant substructures and patterns

Please select ONLY the relevant substructures and patterns for classification:

- [ ] are_connected(V0, V1)
- [ ] triangle(V0, V1, V2)
- [ ] triangle_path(V0, V1, V2)
- [x] direct_line_3(V0, V1, V2)
- [ ] direct_path_3(V0, V1, V2)
- [ ] direct_line_4(V0, V1, V2, V3)
- [ ] direct_path_4(V0, V1, V2, V3)
- [x] y_shape(V0, V1, V2, V3)
- [ ] y_path(V0, V1, V2, V3)
- [ ] y_triangle(V0, V1, V2, V3)
- [ ] y_triangle_path(V0, V1, V2, V3)

Figure 8.4: Choosing relevant patterns/substructures

Also, *classilasp* needs to know what the labels (i.e. class names) are. In our case, there are three labels: **parallel**, **series** and **parallel_series**:

```
Please specify the number of labels (at least 2): 3
* Label 1: parallel
* Label 2: series
* Label 3: parallel_series

 --------------------------------------------------
| Thank you, classification process will now begin. |
 --------------------------------------------------
```

Figure 8.5: Complete pre-processing output

Please make sure the labels start with lower-case letters, and only contain characters from a-z, A-Z, 0-9, as well as '-' and '_'. Ideally, they also shouldn't coincide with any component names. This is the last step before going in the learning cycle.

3. We are prompted to choose whether we want to receive a random model to classify, or one based on a set of constraints we need to write ourselves (i.e. a query). We choose the former for now:

```
(1) Generate random model for classification?
(2) Generate model based on a custom set of constraints?
Your answer: 1
```

Figure 8.6: Choosing type of model for manual classification

This does two things at the same time:

- it pops a diagram image, describing the architecture of the chosen model, as well as its predicted label(s); initially, there are no predicted labels, but these will update with every new set of computed hypotheses:

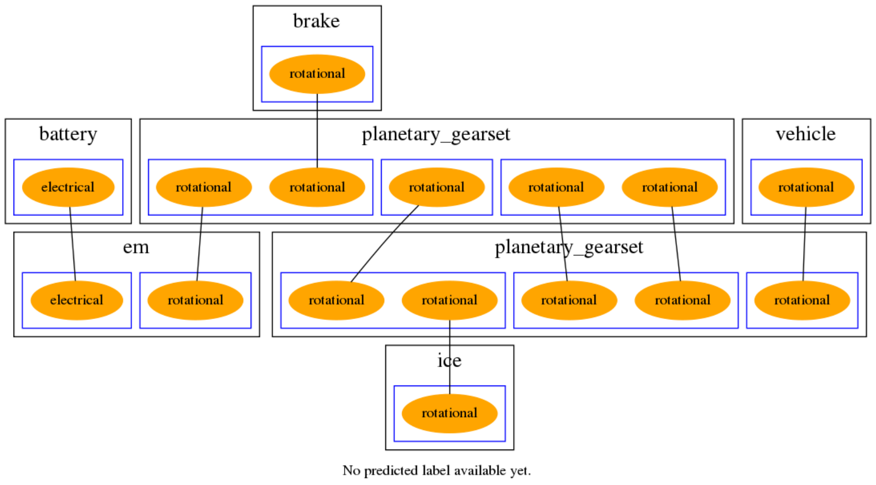


Figure 8.7: Random model for manual classification

- it prints the following in the terminal:

```
* Please classify the model on the screen. Choose (index) from the following:
(1) parallel
(2) series
(3) parallel_series

(0) Skip

Your answer: 1
```

Figure 8.8: Choosing label computed through manual classification

In this case, we say that this model is **parallel**. Note that we could also choose to *skip* the model in case we are not sure; since we are not running this with the ’-n’ flag, a wrong label would lead to wrong hypotheses.

4. After doing this, we are asked the following:

```
Would you like to classify another model? (y/n) y
```

Figure 8.9: Choosing to classify new model or (re)compute hypotheses

Chooosing ’y’ would simply take us back to step 3 in order to classify another model. Choosing ’n’ would prompt *classilasp* to compute hypotheses "explaining" the models classified so far. Obviously, we would choose the former option a few more times, before choosing the latter and get to the following step.

5. This is where we sit back and wait for the hypotheses to be computed. Upon completion, three things happen:

   - the computed hypotheses are printed and the algorithm of choosing random models is recalibrated with them:

```
Would you like to classify another model? (y/n) n

 -------------------------------------------------------------------------------
| Please wait while the hypotheses are being computed, this might take a while. |
 -------------------------------------------------------------------------------

* Hypothesis for label 'parallel':

parallel :- comp(V0, ice), comp(V1, vehicle), direct_line_3(V1, V2, V0).
 --------------------------------------------------

* Hypothesis for label 'series':

 --------------------------------------------------

* Hypothesis for label 'parallel_series':

parallel_series :- comp(V0, battery), direct_line_3(V1, V0, V2).
 --------------------------------------------------

 -----------------------------------------------------------
| Recalibrating algorithm with new hypotheses, please wait. |
 -----------------------------------------------------------

Progress: |██████████████████████████████████████████████████| 100.00%
```

Figure 8.10: Computed hypotheses

   - a pie chart is shown on the screen, displaying the **expected** distribution of labels on the input dataset. Note that these are only approximations and the actual values should be expected to differ slightly:

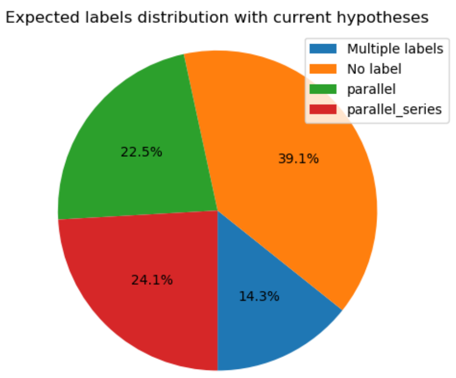


Figure 8.11: Expected labels distribution

   - we get asked the following:

```
Would you like to:
(1) Continue classification to improve current class hypotheses?
(2) Use current hypotheses to automatically classify all initial data?
Your answer (1/2): 1
```

Figure 8.12: Choosing to continue manual classification or do automatic full classification

Given that the current hypotheses predict more than half of the models as either with no label or with multiple labels, we go back to classify more models.

6. Note that this is virtually the same as step 3, but because now *classilasp* knows that there are **definitely** misclassified models (i.e. those with no/multiple predicted labels), it gives the user the option to choose specifically one of those:

```
(1) Generate random model for classification?
(2) Generate model based on a custom set of constraints?
(3) Generate model with no predicted label?
(4) Generate model with multiple predicted labels?
Your answer: 3
```

Figure 8.13: Choosing specifically a model with no predicted models

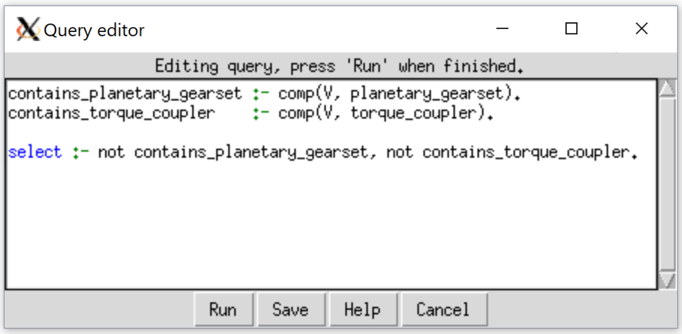
We choose to get a model with (still) no predicted labels. Note that you should always try to prioritize these options, as they are guaranteed to improve the current hypotheses and speed up the overall learning process.

7. Let's say that we know that there are very few **series** models in the input dataset. If we know a certain "configuration" is more likely to produce such models, we can choose option 2 in Step 3, i.e. describe this "configuration" in our own constraints:

```
(1) Generate random model for classification?
(2) Generate model based on a custom set of constraints?
Your answer: 2
```

Figure 8.14: Choosing to query the dataset for specific models

This brings up a text editor where we can write our query. Let's say we know **series** models are more likely to not contain *planetary_gearset* and *torque_coupler* components. We write those in the following way:



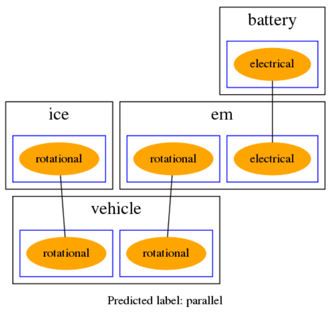
```
contains_planetary_gearset :- comp(V, planetary_gearset).
contains_torque_coupler    :- comp(V, torque_coupler).

select :- not contains_planetary_gearset, not contains_torque_coupler.
```

Figure 8.15: Query Editor

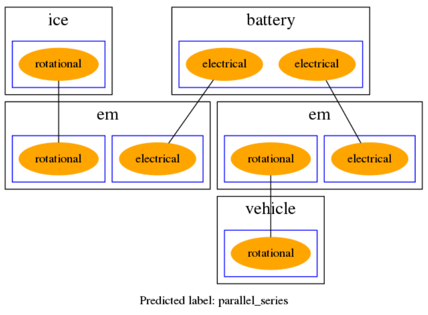
A few things to note here:

- You can press the "Help" button to see what predicates you can use in your query.
- The initial query will be empty unless a different pre-saved one exists (discussed below).
- If your query is valid, it is saved until the next time you wish to use the Query Editor in this run, but will get lost once execution of *classilasp* ends.
- If you wish to save a query as the default **initial** query for all *classilasp* runs, you can click the "Save" button.
- If the text of your query does not change from the last valid query, a model will be chosen from the cache instead of rerunning the query on the entire dataset.

Finally, this query produces the following:



(a) Parallel query result model



(b) Series query result model

Figure 8.16: Query results

We first get the model on the left-hand side, which is not a *series* model, as we would've wanted (it is, as correctly predicted, *parallel*). We choose to *skip* it, then go back to the Query Editor. The previous query is saved there, we simply click "Run" and a new model is chosen from the query cache. This time, we get the model on the right-hand side. This is indeed a *series* model, incorrectly predicted as *parallel_series*.

8. After we classify several models, using the techniques discussed in the previous steps, we can finally choose to automatically classify the full dataset:

```
Would you like to:
(1) Continue classification to improve current class hypotheses?
(2) Use current hypotheses to automatically classify all initial data?
Your answer (1/2): 2

 -----------------------------------------------------------------------
| All initial models are about to be labelled, this might take a while. |
 -----------------------------------------------------------------------

Progress: |████----------------------------------------------| 9.26%
```

Figure 8.17: Full dataset automatic classification

Upon termination, the output is stored in a Prolog file, stored in the same directory as the input file, and its full path is displayed under:

```
* All models have been succesfully labelled and saved in:
```

Figure 8.18: Classified dataset file path

Also, a pie chart displaying the labels distribution over the dataset is shown:

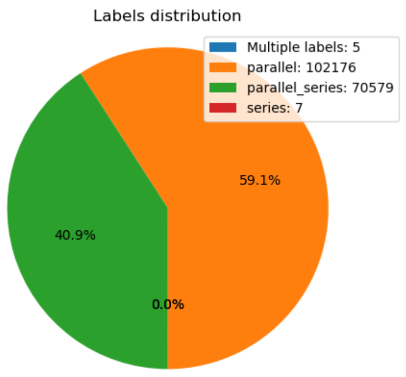


Figure 8.19: Actual labels distribution over the input dataset

Note that execution does not end here; instead, you can now inspect the output file and look for inconsistencies. Also, as is the case with the pie chart from the last step, there may be a few models with no/multiple labels, so you may want to go back to classify some more models (of which you should prioritize those with no/multiple models). In any case, this is displayed as the two options:

```
Would you like to:
(1) Continue classification to improve the class hypotheses?
(2) Exit?
Your answer (1/2): 2
```

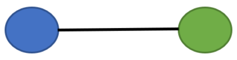
Figure 8.20: Choosing to exit *classilasp* execution

Choosing option 2 simply ends execution and deletes all temporary data (diagram image files, classification programs, etc.), obviously without touching the output file. Note that the actual final hypotheses used for classification are not stored anywhere, so you should manually save them wherever you wish in case you need them.

Appendix A

Appendix

In this section we present all substructures and patterns that are currently defined in *classilasp*. Those contain between 2 and 4 nodes. Before showing them, we need to make the distinction between a direct connection and a basic path (which is an arbitrarily long path between two nodes, containing no cycles):


(a) Direct connection via (single) edge

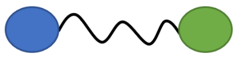

(b) Arbitrarily long basic path

Figure A.1: Direct connection vs basic path notation

Note that, whenever a graph contains multiple basic paths, they should be considered to be mutually disjoint, except perhaps for extremities clearly showed as intersections. Also, all nodes in these graphs represent **distinct** components. Lastly, though somewhat similar in definition, substructures and patterns are different, as will be seen:

- **substructures** involve only direct connections and are thus local connections and are thus local parts of a graph

- **patterns** involve only basic paths and can thus be arbitrarily large, describing general aspects of a whole graph

Combinations of the two notions are indeed possible by simply choosing the relevant patterns and substructures to form them; *classilasp* (through ILASP) can then learn them automatically as part of the hypotheses.

2 nodes

Note that all graphs in the input file are considered to be connected, i.e. there is a path from any node to any other node, so a 2-node pattern would be by default true for any nodes, therefore redundant. However, a local 2-node substructure is needed, as it describes that two specific components are connected:

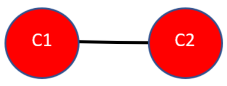
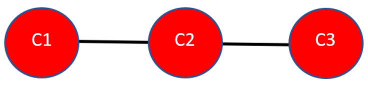
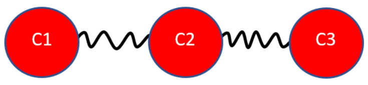


Figure A.2: **are_connected(C1, C2)**

3 nodes

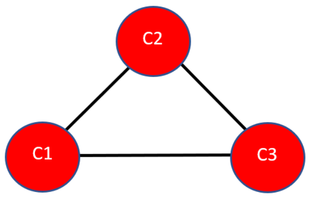


(a) **direct_line_3(C1, C2, C3)**

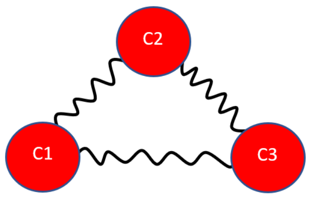


(b) **direct_path_3(C1, C2, C3)**

Figure A.3: direct_line_3 vs direct_path_3



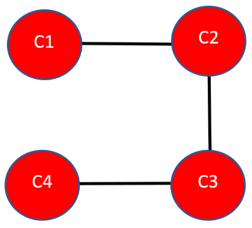
(a) **triangle(C1, C2, C3)**



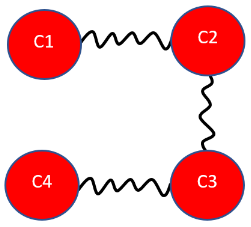
(b) **triangle_path(C1, C2, C3)**

Figure A.4: triangle vs triangle_path

4 nodes

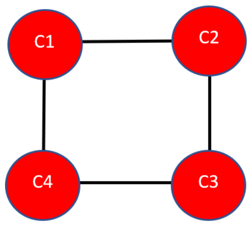


(a) **direct_line_3(C1, C2, C3, C4)**

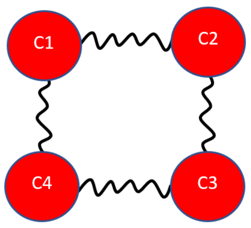


(b) **direct_path_3(C1, C2, C3, C4)**

Figure A.5: direct_line_4 vs direct_path_4

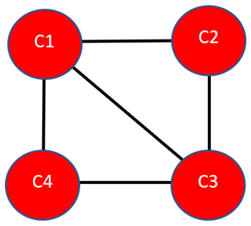


(a) **square(C1, C2, C3, C4)**

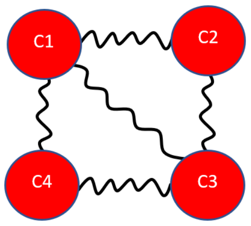


(b) **square_path(C1, C2, C3, C4)**

Figure A.6: square vs square_path

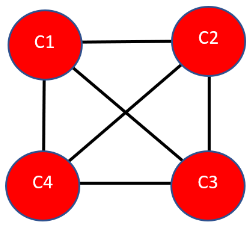


(a) **square_one_diag(C1, C2, C3, C4)**

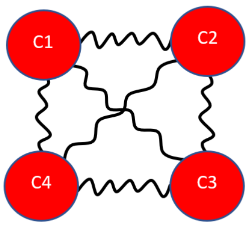


(b) **square_one_diag_path(C1, C2, C3, C4)**

Figure A.7: square_one_diag vs square_one_diag_path

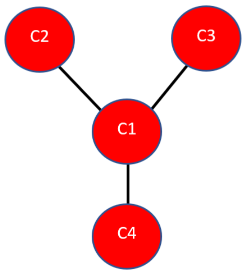


(a) **connected_4(C1, C2, C3, C4)**

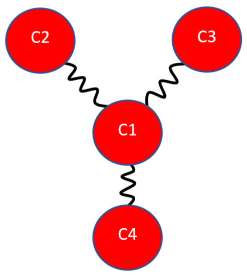


(b) **connected_path_4(C1, C2, C3, C4)**

Figure A.8: connected_4 vs connected_path_4

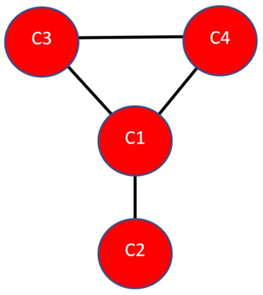


(a) **y_shape(C1, C2, C3, C4)**

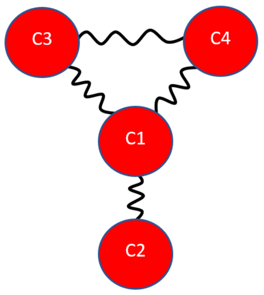


(b) **y_path(C1, C2, C3, C4)**

Figure A.9: y_shape vs y_path



(a) **y_triangle(C1, C2, C3, C4)**



(b) **y_triangle_path(C1, C2, C3, C4)**

Figure A.10: y_triangle vs y_triangle_path

Bibliography

[1] The basic rc filters. http://www.experimentalistsanonymous.com/ve3wwg/doku.php?id=rc_simple_filter. Accessed: 10-06-2018.

[2] Classes of organic compounds. https://courses.lumenlearning.com/boundless-chemistry/chapter/classes-of-organic-compounds/. Accessed: 10-06-2018.

[3] graphviz. https://pypi.org/project/graphviz/. Accessed: 21-04-2018.

[4] Inductive logic programming. https://en.wikipedia.org/wiki/Inductive_logic_programming. Accessed: 23-01-2018.

[5] Mean shift cluster analysis example with python and scikit-learns. https://pythonprogramming.net/hierarchical-clustering-machine-learning-python-scikit-learn/. Accessed: 21-04-2018.

[6] Mean shift scalability. http://scikit-learn.org/stable/modules/generated/sklearn.cluster.MeanShift.html. Accessed: 21-04-2018.

[7] Meanshift algorithm for the rest of us. http://www.chioka.in/meanshift-algorithm-for-the-rest-of-us-python/. Accessed: 21-04-2018.

[8] Redux. https://redux.js.org/introduction. Accessed: 07-06-2018.

[9] Siemens. https://en.wikipedia.org/wiki/Siemens. Accessed: 10-01-2018.

[10] *Electric vehicles in Europe*. https://www.eea.europa.eu/publications/electric-vehicles-in-europe. Accessed: 10-01-2018.

[11] Tkinter. https://wiki.python.org/moin/TkInter. Accessed: 07-06-2018.

[12] Xming. https://xming.en.softonic.com/. Accessed: 10-01-2018.

[13] C. Anger, K. Kongczak, T. Linke, and T. Schaub. *A glimpse of answer set programming. Künstliche Intelligenz*, 19(1):12–17, 2005.

[14] M. Arias, R. Khardon, and J. Maloberti. *Learning Horn Expressions with LOGAN-H. Journal of Machine Learning Research*, 8:549–587, 2007.

[15] C. Bessiere, R. Coletta, E. Hebrard, G. Ketsirelos, N. Lazaar, N. Narodytska, C.G. Quimper, and T. Walsh. *Constraint Acquisition via Partial Queries. Proceedings of the Twenty-Third International Joint Conference on Artificial Intelligence*, August 2013.

[16] J. Cagan, M.I. Campbell, S. Finger, and T. Tomiyama. *A Framework for Computational Design Synthesis: Model and Applications. Journal of Computing and Information in Engineering, ASME*, 5:171–181, September 2005.

[17] Dorin Comaniciu and Peter Meer. Mean shift: A robust approach toward feature space analysis. *IEEE Transactions on Pattern Analysis and Machine Intelligence*, 24:603–619, 2002.

[18] Abderrazak Daoudi. *Learning and Using Structures for Constraint Acquisition*. PhD thesis, FSR, Universite Mohammed V, Maroc, May 2016.

[19] T. Eiter, G. Ianni, and T. Krennwallner. *Answer Set Programming: A Primer*. http://www.kr.tuwien.ac.at/staff/tkren/pub/2009/rw2009-asp.pdf, 2009.

[20] Andrea Fuksova. *Fast Relational Learning Using Bounded LGG*. Master's thesis, Department of Computer Science, Czech Technical University in Prague, 2014.

[21] M. Gebser, R. Kaminski, B. Kaufmann, M. Lindauer, M. Ostrowski, J. Romero, T. Schaub, and S. Thiele. *Potassco: User Guide*. 2017.

[22] M. Gebser, R. Kaminski, B. Kaufmann, M. Ostrowski, T. Schaub, and M. Schneider. *Potassco: The Potsdam Answer Set Solving Collection*. 24(2):107–124, 2011.

[23] H. Kries, R. Bloomberg, and D. Hilvert. *De novo enzymes by computational design*. *Current Opinion in Chemical Biology*, 17:221–228, 2013.

[24] N. Landwehr, K. Kersting, and L. De Raedt. *Integrating Naive Bayes and FOIL*. *Journal of Machine Learning Research*, 8:481–507, 2007. www.jmlr.org/papers/volume8/landwehr07a/landwehr07a.pdf. Accessed: 25-01-2018.

[25] Mark Law, Alessandra Russo, and Krysia Broda. *Learning Weak Constraints in Answer Set Programming*. https://spiral.imperial.ac.uk/bitstream/10044/1/33615/8/AcceptedVersion.pdf. Accessed: 19-01-2018.

[26] Mark Law, Alessandra Russo, and Krysia Broda. *Inductive Learning of Answer Set Programs*. *Fermé E., Leite J. (eds) Logics in Artificial Intelligence. JELIA 2014. Lecture Notes in Computer Science, vol 8761. Springer, Cham*, 2014.

[27] Mark Law, Alessandra Russo, and Krysia Broda. *Inductive Learning of Answer Set Programs*. https://www.doc.ic.ac.uk/~ml1909/ILASP, 2015.

[28] E.A. Lee, J. Roychowdhury, and S.A. Seshia. *Fundamental Algorithms for System Modeling, Analysis, and Optimization*. *U.C.Berkeley*, 2011. https://embedded.eecs.berkeley.edu/eecsx44/fall2011/lectures/SATSolving.pdf. Accessed: 18-01-2018.

[29] H. Lipson and J.B. Pollack. *Automatic design and manufacture of robotic lifeforms*. *Nature*, 406:974–978, 2000.

[30] S. Muggleton, J. Santos, and A. Tamaddoni-Nezhad. *ProGolem: A System Based on Relative Minimal Generalisation* . *Department of Computing, Imperial College London*, 2009. http://ilp.doc.ic.ac.uk/ProGolem/progolem.pdf. Accessed: 25-01-2018.

[31] F. Pedregosa, G. Varoquaux, A. Gramfort, V. Michel, B. Thirion, O. Grisel, M. Blondel, P. Prettenhofer, R. Weiss, V. Dubourg, J. Vanderplas, A. Passos, D. Cournapeau, M. Brucher, M. Perrot, and E. Duchesnay. Scikit-learn: Machine learning in Python. *Journal of Machine Learning Research*, 12:2825–2830, 2011.

[32] H.K. Privett, G. Kiss, T.M. Lee, R. Bloomberg, R.A. Chica, L.M. Thomas, D. Hilvert, K.N. Houk, and S.L. Mayo. *Iterative approach to computational enzyme design*. *PNAS*, 109:3790–3795, 2011.

[33] A. Russo, M. Law, and K. Broda. *Inductive Learning of Human Behaviours*. *Department of Computing, Imperial College London*, 2016. http://mi20-hlc.doc.ic.ac.uk/short_presentations/Russo.pdf. Accessed: 25-01-2018.

[34] Alessandra Russo, Mark Law, and Stephen Muggleton. *Logic-based Learning (C304) course notes, Imperial College London*.

[35] Marek Sergot. *Knowledge Representation - Stratified logic programs*. https://www.doc.ic.ac.uk/~mjs/teaching/KnowledgeRep491/Stratified_491-2x1.pdf, January 2005.

[36] Ashwin Srinivasan. Aleph. www.cs.ox.ac.uk/activities/machinelearning/Aleph/aleph. Accessed: 23-01-2018.

[37] Lappoon R. Tang. *Integrating Top-down and Bottom-up Approaches in Inductive Logic Programming: Applications in Natural Language Processing and Relational Data Mining*. PhD thesis, Department of Computer Sciences, University of Texas, Austin, TX, August 2003.

[38] James Worell. *The DPLL Algorithm*. *Logic and Proof, Oxford*, 2016. http://www.cs.ox.ac.uk/james.worrell/lec7-2015.pdf. Accessed: 15-01-2018.