Fundamenta Informaticae XXI (2001) 1001–1032
IOS Press

A Dichotomic Search Algorithm for Mining and Learning in Domain-Specific Logics

Sébastien Ferré[*] [C]
Irisa, Université de Rennes 1
Campus de Beaulieu, 35042 Rennes cedex, France
Email: ferre@irisa.fr

Ross D. King
Department of Computer Science
University of Wales, Aberystwyth
Aberystwyth, SY23 3DB, Wales, UK
Email: rdk@aber.ac.uk

Abstract. Many application domains make use of specific data structures such as sequences and graphs to represent knowledge. These data structures are ill-fitted to the standard representations used in machine learning and data-mining algorithms: propositional representations are not expressive enough, and first order ones are not efficient enough. In order to efficiently represent and reason on these data structures, and the complex patterns that are related to them, we use *domain-specific logics*. We show these logics can be built by the composition of logical components that model elementary data structures. The standard strategies of top-down and bottom-up search are ill-suited to some of these logics, and lack flexibility. We therefore introduce a *dichotomic search* strategy, that is analogous to a dichotomic search in an ordered array. We prove this provides more flexibility in the search, while retaining completeness and non-redundancy. We present a novel algorithm for learning using domain specific logics and dichotomic search, and analyse its complexity. We also describe two applications which illustrates the search for motifs in sequences; where these motifs have arbitrary length and length-constrained gaps. In the first application sequences represent the trains of the East-West challenge; in the second application they represent the secondary structure of Yeast proteins for the discrimination of their biological functions.

[*]BBSRC grant 21BEP17028
[C]Corresponding author

Keywords: Data structures, domain-specific logics, search algorithm, dichotomy, formal concept analysis, concept learning, pattern mining, bioinformatics.

1. Introduction

Propositional representations, like valued attributes and item-sets, have been widely used in data-mining and machine learning [35]. Although they are often highly efficient, these representations are known to lack the expressivity needed in many applications where structured data is important: e.g., bioinformatics, and text mining. The related research areas of Inductive Logic Programming (ILP) [38] and relational data-mining [13] are one possible solution to this problem, as these methods allow for the search of complex relational patterns and rules. However, the generality of relational representations based on Predicate Logic, makes it difficult to match specific needs of applications in an efficient way. So, there is a need for intermediate representation languages to provide better trade-offs between expressivity and efficiency. One example of this is in the representation of sequences for the discovery of motifs [33, 32, 2], with the important special case of biological sequences [10, 9]. Other common data structures are graphs (e.g., chemical molecules [14]) and trees (e.g., document structures [1]).

We propose to use *domain-specific logics* as intermediates between propositional logic and predicate logic. The term *logic* is here used as a shorthand for a search space defined by a representation language ordered by a generalization relation (alias *subsumption*). A key issue with domain-specific logics is how generic the search algorithm is, as the search space may change from one application to another. This approach therefore requires generic programming techniques for separating the search algorithm from logic-specific operations (e.g., refinement operators). We further use the technology of *logic functors* [18] that enables to build logics from simpler parts. Each logic functor corresponds to a data structure (e.g., interval, sequence), and the definition of logics becomes similar to the definition of complex types from primitive types in programming languages. Section 2 motivates and defines domain-specific logics, and illustrates them by a detailed example.

In classification machine learning tasks the objective is to discriminate target concepts by some computable function (e.g. rules). Most existing algorithms search for them by traversing the search space in a top-down or bottom-up fashion. We propose a new search algorithm that focuses on the concepts to be discriminated, rather than hypotheses themselves (*concept-based search*), and which explores the search space in both direction with bigger leaps (*dichotomic search*). The first advantage of this is that the search is more data-driven, and can cope with infinite chains in the search space, allowing a wider range of logics to be used. Secondly, it combines completeness (w.r.t. concepts), non-redundancy, and flexibility at the same time, which has been recognized as a difficult problem [5]. This available flexibility allows more heuristics to be used to guide the search. Section 3 presents the principles of this concept-based and dichotomic search, and Section 4 develops theoretical foundations, an algorithm and its complexity.

Our search algorithm relies on a new logical operation: *dichotomy* in a *logical interval*. In Section 5, we show how it can be defined for a few data structures. Section 6 presents experiments in 2 applications: (1) the East-West train problem as an artificial example, where trains are represented as car sequences, and (2) protein function predictions from biological sequences as a real-world application.

Finally, we discuss limits and perspectives of the concept-based and dichotomic search in Sections 7 and 8.

2. Logical Search Spaces

We use *logic* as a generic term for denoting a symbolic representation language for hypothesis search space in learning tasks. A logic is defined as a language of formulas, which are partially ordered by a generalization ordering, often called *subsumption*. Given a dataset and a target concept, each formula can be evaluated as a hypothesis of the target concept with measures like support or accuracy. The task of learning the target concept consists in searching formulas that maximize such a measure. The main criteria for choosing a logic are the expressivity and relevance of the language, and the efficiency of searching in it. We first discuss the two most common logics (propositional logic and predicate logic) w.r.t. these criteria, and then motivate our choice for *domain-specific logics* (DSL).

2.1. Propositional Logic

Most machine learning methods relies on propositional representations [35]. In this setting hypotheses are combinations of predefined (valued) attributes or features. Combinations can be symbolic (e.g., item-sets, decision trees), or numerical (e.g., neural nets, regression).

Learning in propositional languages is usually efficient, but unless the chosen attributes fit the problem well they often lack expressivity to find good predictive functions. Indeed, not every learning problem can be correctly represented propositionally with a reasonable number of attributes, and Inductive Logic Programming (ILP, [38]) has been introduced to deal with relational problems. A well known example of a relational problem is the East-West train challenge [34], originally initiated by Michalski. The 10 original trains are depicted in Figure 1 with East-bound trains on the left, and West-bound trains on the right.

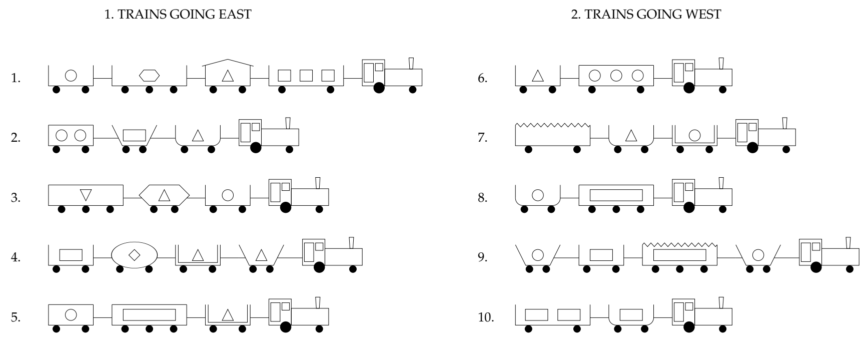

Figure 1. The 10 original Michalski's trains.

In these trains, each car can be described in a propositional way by attributes such as length, shape, number of wheels, shape of the roof, and number and shape of loads. However, trains are sequences of cars. A natural propositionalization would be to duplicate the car attributes and index them according to the car position in the train. But this assumes a bounded size on the trains, as the set of attributes must be fixed *a priori*; and the ability to handle missing values in shorter trains. Also, this makes it impossible to express simple patterns like “some car is short and has a roof” because all attributes are position-specific. Another approach would be to make this pattern a Boolean attribute; but in general the number of all possible patterns is far too large (or even infinite) to allow for a complete propositionalization.

2.2. Predicate Logic

The principle of Inductive Logic Programming (ILP) is to generalize propositional learning methods by replacing the propositional logic by Predicate Logic (PL), or at least, some fragment of PL. PL allows the representation of relations and variables. The fragments of PL used are usually close to Prolog (Horn clauses) or Datalog (Prolog without functions). This enables very expressive logics — if we consider that Prolog is a full programming language.

```
has-car(t2,c21).          has-car(t2,c22).          has-car(t2,c23).
length(c21,short).        length(c22,short).        length(c23,short).
axes(c21,2).              axes(c22,2).              axes(c23,2).
shape(c21,rectangle).     shape(c22,bucket).        shape(c23,ushape).
double(c21,no).           double(c22,no).           double(c23,no).
roof(c21,flat).           roof(c22,none).           roof(c23,none).
loads(c21,2).             loads(c22,1).             loads(c23,1).
lshape(c21,circle).       lshape(c22,rectangle).    lshape(c23,triangle).
next-car(c21,c22).        next-car(c22,c23).
```

Table 1. Description of the 2nd Michalski’s train by a set of relational facts.

In the train example, all the initial information can be expressed in a relational representation of the whole trains. For instance, the second train `t2` can be represented by the set of relational facts in Table 1, where `c21` denotes the first car and so on. Then, the pattern “some car is short and has a roof” can be expressed by the clause

```
has-car(T,C), length(C,short), not roof(C,none),
```

where `T` and `C` are variables standing respectively for some train and some car.

In fact, PL alone is not enough to provide expressiveness, given that existing algorithms cannot rely on predicate invention. What characterizes the expressivity of PL in ILP is the background knowledge, which is different from one application to another. This Background Knowledge (BK) is made of:

- facts adding information about examples,
- rules defining intentional predicates useful to the expression of hypotheses,
- various constraints like the type and mode of predicates, or a minimum support for hypotheses.

The BK contributes to defining the language bias. The introduction of new predicates increases the expressivity, while the introduction of constraints typically decreases it. So, in some sense, rather than being a very expressive logic for learning, PL is more a framework for defining various language bias, and applying to them a common learning algorithm. This generic approach is certainly flexible and powerful, but its one-language solution has also some drawbacks w.r.t. efficiency, as we now explain.

2.3. Genericity vs Efficiency

Because Predicate Logic is so generic and expressive, it tends to be less efficient than a specific logic when applied to a fragment of PL. For instance, it is notorious that ILP is very poor at managing numerical values. We illustrate this with the expression of numerical intervals on the East-West train problem, but this problem is general enough to apply to different kinds of patterns, like sub-sequences. In the above relational representation of trains, the predicate `loads/2` represents the number of loads of each car. To enhance the expressivity of patterns we add the predicate `loads/3` representing an interval on the number of loads: for example, `loads(C,1,3)` means the number of loads of some car is comprised between 1 and 3. We also add some Background Knowledge (BK) to define this new predicate and relate it to the facts describing the trains:

```
loads(C,Min,Max) :- loads(C,V), Min <= V, V <= Max.
loads(C,Min,Max) :- loads(C,Min1,Max1), Min <= Min1, Max1 <= Max.
```

Then, it is possible to prove the following logical implications ($\models$):

`loads(C,2)` $\models$ `loads(C,1,3)` $\models$ `loads(C,1,5)`.

However, this logical implication is computationally costly, and even undecidable in the general case. For this reason, most ILP algorithms rely on a weaker implication called θ-subsumption and denoted by $\sqsubseteq_\theta$. It is a weaker implication in the sense that it is not complete, i.e. there are valid implications that are not valid θ-subsumptions. For instance, none of the above implications hold with subsumption.

This incompleteness of subsumption has several unfortunate consequences. Firstly, it is a cause for redundancy in the search for patterns because: (1) some patterns are not recognized as logically equivalent, and (2) if the pattern `loads(C,1,3)` is recognized as a stop in a top-down search, all its sub-intervals will still be visited because they are not recognized as subsumed by `loads(C,1,3)`. Secondly, in some algorithms (e.g., Aleph [45], Golem [37]), it leads to problems with the construction of a *bottom-clause* (which is roughly the set of all ground facts that can be derived about an example) [36]. In the above example, the bottom clause consists of all possible intervals containing the value given by the predicate `loads/2`. In the general case, this bottom clause can be exponentially large, or even infinite in the case of general first-order horn-clause logic [41]. So, the efficiency that is saved by using subsumption is balanced by redundancy in the search, and the cost of computing the bottom clause.

Another argument for using specific logics is the importance of removing as much irrelevant hypotheses as possible from the search space, as this heavily determines the efficiency of the search. The restriction of the language of possible hypotheses is specified by a language bias, which express syntactical constraints: e.g. predicate types and modes, relational clichés. A limitation is that no bias language can remove all irrelevant hypotheses unless it is a full programming language. In the latter case, it is possible to define a new language from scratch, which means the advantage of genericity is lost. The conclusion is that a unique logic is not compatible with efficiency, unless it matches closely the desired language of hypotheses.

We now give a few examples of simple patterns that would be useful in the train application, and are not naturally handled by ILP.

Numerical values and intervals. This can be applied to the number of wheels or the number of loads of each car. The usual approach is to discretize the possible values in a finite set of disjoint intervals. The more expressive solution of considering arbitrary intervals can easily become intractable.

Taxonomies of terms. This can be applied to the shapes of cars, roofs, and loads. For instance, a rectangle and a triangle could be generalized as a polygon. The usual approach is to flatten the taxonomy by describing a load for instance by both shapes rectangle and polygone. A consequence is that the taxonomic ordering is lost in the hypotheses search space.

Global properties. This can be applied to train properties like its length (number of cars), or total number of loads. An approach is to add explicit information in the description of trains. Another approach is to define new predicates in the background knowledge, but this leads also to add explicit information if the bottom clause has to be computed.

All these approaches introduce redundancy in the search, because the subsumption does not reflect the natural ordering (e.g., between intervals or shapes), and also intractability by the necessity to make explicit a lot of implicit information.

2.4. Domain-Specific Logics

It can be observed that, paradoxically, what appears difficult to represent in ILP could very easily be represented by *Domain-Specific Logics* (DSL). For instance, it is easy to define a logic of numerical intervals with a natural subsumption ordering. The same for taxonomies of terms. More generally, DSLs make it possible to design efficient representations and subsumption relations for various data structures (e.g., sequences, trees, graphs), and for various application domains (e.g., bioinformatics, natural language processing). DSLs are efficient because: (1) the language can be efficiently restricted to relevant hypotheses, (2) the subsumption can most often be made complete, and so (3) no implicit information needs to be explicited.

This specialization approach has already been studied, especially for structures like graphs [25, 44], trees [1], and sequences [33, 32, 2]; and for instance applied to the representation of chemical molecules [14, 46], XML documents [1], and biological sequences [11] or command sequences [32]. In the latter example, a fragment of predicate logic is used along with a specific refinement operator in the ILP framework. This makes it possible to discover sequence patterns in a more efficient way than would be possible with standard ILP. This approach is not specific to machine learning and data-mining, and can also be found in other domains such as Constraint Logic Programming (CLP) [21], and kernel-based Support Vector Machines (SVM) [31]. What is new here is that we make this approach systematic, and provide means to application developers for building fresh DSLs from existing parts (see next section).

In fact, DSLs are not so much an alternative approach to ILP than an abstraction of it. Indeed, the logic used in ILP can be made a DSL that is parameterized by some background knowledge. So, instead of a general-purpose logic (predicate logic) and a search algorithm specific to this logic, we propose to use a generic search algorithm working on an abstract logic. In this case DSLs are developed according to specific needs and pluged in this generic search algorithm. A similar approach has been applied in program analysis [3], and logical information systems [18, 19].

Definition 2.1. (logic and subsumption)
A *(domain-specific) logic* is defined as a couple $\mathcal{L} = (L, \sqsubseteq)$, where L is a (possibly infinite) language of formulas, and $\sqsubseteq$ is a generalization ordering on formulas called *subsumption*. Given 2 formulas $f, g \in L$, $f \sqsubseteq g$ denotes that g subsumes f (or f is subsumed by g). When $f \sqsubseteq g$ and $g \sqsubseteq f$, we say that f and g are logically equivalent, and we note $f \equiv g$.

Let $F \subseteq L, g \in L$. The notation $F \sqsubseteq g$ is equivalent to $\exists f \in F : f \sqsubseteq g$; and the notation $g \sqsubseteq F$ is equivalent to $\forall f \in F : g \sqsubseteq f$.

A semantics is often associated to a logic, and helps to base the consistency and completeness of the subsumption relation. Consistency is required in order to avoid misclassifi cation of examples, or misordering of hypotheses in the search space. Completeness is also required between examples and hypotheses, once again to avoid misclassifi cation, but not necessarily between hypotheses. This incompleteness can result in a loss of effi ciency in the search, but also allows for more expressive logics in which subsumption is not fully decidable (e.g., predicate logic). The reason for this asymetric requirement is that logical subsumption entails coverage containment over a given dataset (what can be used for pruning the search), but not the reverse. So, we state the following hypothesis for the remaining of this paper.

Assumption 2.1. (subsumption)
In a DSL $\mathcal{L} = (L, \sqsubseteq)$, we assume that the subsumption $\sqsubseteq$ is consistent, and complete when the left hand-side is an example description, but not necessarily complete in other cases.

Like PL that can be customized by the specifi cation of background knowledge, DSLs can be made customizable. For instance, a DSL about sequences can be given parameters like the maximal length of sub-sequences, and the possibility to use wildcards or gaps. This means that DSLs can cover a very wide range of logics, from very specifi c and hard-coded logics, to very general and fexible logics.

2.5. Generic Programming with Logic Functors

The advantage of using a unique representation language, say Predicate Logic, is that there is very little to do in order to apply a learning algorithm to new applications. This contrasts with DSLs where it could be necessary to build a brand new logic for every new application. This is clearly undesirable as defi ning and implementing a logic requires both logic and programming skills. Fortunately, the technology of *logic functors* [18] makes it possible to build logics by simple composition of existing parts. These parts, the logic functors, are logical components that encapsulate both representation and reasoning capabilities for concrete domains (e.g., integers, strings), data structures (e.g., sets, sequences), and general combiners (e.g., sum and product of logics). We gather these functors into a library (which already contains about 20 reusable components).

2.5.1. Sequences with Complex Elements and Flexible Gaps

Figure 2 illustrates the defi nition of a DSL for the train dataset. Each sub-tree defi nes a sub-logic, and each node is a functor that is applied to one or several sub-logics. For instance, the functor `Interv` can represent intervals over any domain of values that is totally ordered (e.g., integers, dates, times). So, it is applied to the concrete domain `Int` in order to represent intervals over integers. The same mechanism applies to all non-terminal functors. We now give a short description of each functor, as well as logical formulas they allow to express.

`Attr`: simple attribute names. The set of allowed names can be given as parameters; otherwise any name is allowed. Taxonomic relations can be defi ned over attributes. For instance, in the train example, the attribute `Roof` is made more general than any roof shape (e.g., `FlatRoof`, `PeakedRoof`).

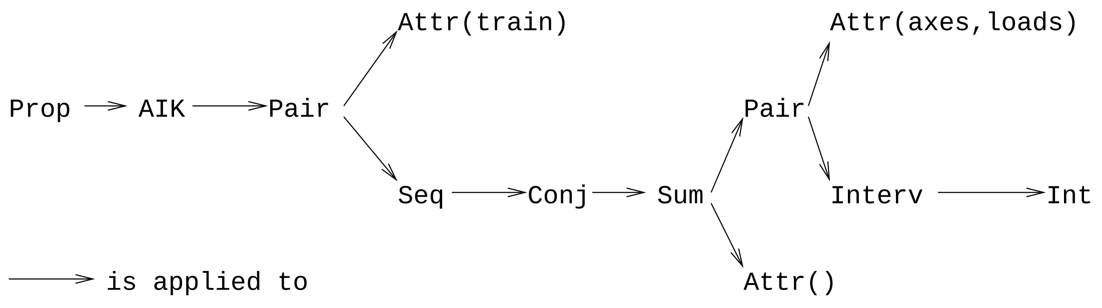

Figure 2. The definition of a DSL for the train dataset by composition of logic functors.

`Int`: integer values.

`Interv(Val)`: intervals overs elements of the sub-logic `Val`. Intervals can be open to the right and/or to the left. Syntax: /`=Val`/, /`<=Val`/, /`>=Val`/, /`in Val..Val`/.

`Pair(L1,L2)`: product of the 2 sub-logics `L1` and `L2`. This enables to build pair of formulas, e.g. valued attributes from attribute names and value domains. Syntax: /`L1 L2`/ (concatenation).

`Sum(L1,L2)`: sum of the 2 sub-logics `L1` and `L2`. This enables to combine different kinds of data representations. Syntax: /`L1`/, /`L2`/.

`Conj(Atom)`: conjunctions (or item-sets) over elements of the sub-logic `Atom`. Syntax: /`Atom & ...& Atom`/.

`Seq(Elt)`: sequence motifs, where the sequence elements are defined by the sub-logic `Elt`. This enables a taxonomy on the sequence elements. Motifs can contain flexible gaps. Syntax: /`match ( X - ...- X )`/, where `X` can be either `Elt` (an element) or `x(Min,Max)` (a gap covering between `Min` and `Max` elements).

`AIK(L)`: epistemic layer over the sub-logic `L` in order to apply the Closed World Assumption on example descriptions. This epistemic layer is necessary for expressing negative features. Syntax: /`[L]`/ (a description), /`L`/ (a feature).

`Prop(Atom)`: closure of the sub-logic `Atom` by the 3 boolean operators (`and`, `or`, `not`).

This may seem a very detailed decomposition of a logic that is relatively simple after all: i.e., sequences of conjunctions of simple and valued attributes. But the more detailed this decomposition is, the more reuseable the logic functors are.

As an illustration of the above logic, we translate a description and a few hypotheses `h` that can be expressed (a translation in PL is also given for comparison):

1. description of the 2nd train:
   (DSL) `d = [train match (`
   `Rectangle & Short & NotDouble & FlatRoof & axes=2 & LCircle & loads=2 -`
   `Bucket & Short & NotDouble & NoRoof & axes=2 & LRectangle & loads=1 -`

`UShaped & Short & NotDouble & NoRoof & axes=2 & LTriangle & loads=1 ) ]`
(PL) see Table 1

2. hypothesis “trains containing a short roofed car”:
   (DSL) `h <- train match ( Short & Roof )`
   (PL) `h(T) <- has-car(T,C), length(C,short), not roof(C,none)`

3. hyothesis “trains containing a short car before a roofed car with 1 or 2 other cars in between”:
   (DSL) `h <- train match ( Short - x(1,2) - Roof )`
   (PL) `h(T) <- has-car(T,C), length(C,short), next(C,C1), next(C1,C2),`
   `(C2 = C3 ; next(C2,C3)), not roof(C3,none)`

4. hypothesis “trains containing a short car with at least 2 loads, and no roofed car”:
   (DSL) `h <- train match ( Short & loads >= 2 and not train match Roof )`
   (PL) `h(T) <- has-car(T,C1), length(C1,short), loads(C1,N), N >= 2,`
   `not (has-car(T,C2), not roof(C2,none))`

This shows another advantage of DSLs: a more natural and more compact syntax. A difference with PL in this example is that variables are implicit, like in Description Logics [8]. But this is not a general observation as DSLs can perfectly use variables in their languages if necessary.

2.5.2. Trees and Graphs

Given the ability to define recursive logic functors[1], a logic functor `Tree` depending on a logic `Node` used to represent its nodes can be defined by composing the logic functors defined above. No additional component is necessary:

```
Tree(X,Node) = Sum(Node,Pair(Node,X(Tree(X,Node)))),
```

i.e., a tree is either a node, or a pair made of a node and some combination of subtrees.

The type of this combination is determined by the logic functor `X`, passed as a first argument to `Tree`. In the case of a binary tree, it can be defined as `X(T) = Pair(T,T)`. In the case one is interested in representing XML documents and wants to express sequence patterns over a set of subtrees, it can simply be defined by `X(T) = Seq(T)`. For instance, the logic `Tree(Seq,Atom)` allow to express the tree pattern

```
document match (section match (text - picture) - x(0,1) - section match
(text - picture)),
```

which denotes a document containing two sections separated by at most one element, each section containing a text followed by a picture.

Unlike trees, and like sequences, the representation of graphs requires the definition of a new logic functor `Graph(Node,Edge)`, parameterized by 2 sub-logics: one for the representation of nodes, the other for the representation of edges. However, this is out of the scope of this paper, and we focus on sequences in the remaining.

[1]Recursive functors are for instance provided by the functional programming language Objective Caml from version 3.07.

2.6. Learning Tasks

The most common task in machine learning is classifi cation (concept learning, discrimination). Given a logic $(L, \sqsubseteq)$, and a target concept c that discriminates between positive examples E^+ and negative examples E^-, the objective is to fi nd a hypothesis $h \in L$ that discriminates it [35]. This discrimination is based on the subsumption test $\sqsubseteq$ between this hypothesis and each example. Ideally, h correctly discriminates the target concept c if h subsumes all positive examples ($\forall e \in E^+ : e \sqsubseteq h$), and subsumes no negative example ($\forall e \in E^- : e \not\sqsubseteq h$). In practice, “all/no” is often replaced by “nearly all/no” in order to take into account noise in the data.

This search for a hypothesis can be reduced to the search for *rules*. The difference is that a rule must be *consistent* (covering no negative example), but not necessary *complete* (covering all positive examples). A complete hypothesis can be built by disjuncting several incomplete rules. A common strategy to achieve this is separate-and-conquer [22]. The advantage of this approach is to have an expressive hypothesis language (allowing disjunction), while having a reduced search space of rules (without disjunction).

The search for rules can be further reduced to the search for *features*, where a feature needs to be neither consistent, nor complete. A consistent rule can be built by conjuncting features and negations of features. A common strategy is to search for all frequent features (or patterns) in a given feature language [14, 2, 27], and then to apply a propositional learner in order to produce a set of rules, i.e. a hypothesis.

To summarize, there are 3 different possible tasks when searching in logical spaces: (1) search for hypotheses (both consistent and complete), (2) search for rules (consistent, but not necessary complete), and (3) search for features (not necessary consistent, and not necessary complete). Of course, rules and features covering more positive examples, and features covering less negative examples should be prefered, so that there is a graduation between these different tasks. Hypotheses/rules/features are all logical formulas, but we use the former words when we want to emphasize their relationship to a dataset.

3. Concept-Based and Dichotomic Search

The logic chosen as a representation language for features, rules or hypotheses forms a search space in which some elements have to be found according to some criteria. The elements of the search space are the logical formulas, and the generalization ordering is realized by the subsumption relation (cf. Defi nition 2.1). The criteria to be satisfi ed depends on the learning task. In the inductive concept learning task, this criteria is roughly related to the accuracy and coverage of an hypothesis w.r.t. a target concept. In the feature mining task, this criteria is less constrained and is roughly the power of a feature to discriminate the target concept (e.g., information gain).

According to Fürnkranz [22], the search bias can be split in the search algorithm, the search strategy, and the search heuristics. After a short discussion about these different aspects in Section 3.1, we introduce the principles for a new search strategy: a *concept-based* and *dichotomic* search.

3.1. Discussion on the Search Bias

Most search biases fall in one of the two search strategies: *top-down* search or *bottom-up* search. In the most common of them, top-down search, the search space is explored by starting from the most general formula covering all examples (e.g., the empty sub-sequence), and iteratively specializing it. In concept learning, the specialization usually stops when a formula covers no more negative examples. In feature generation, it can stop when the formula's coverage comes under a frequency threshold [13]. In some systems a *bottom formula* is used to guide the search and to ensure that the coverage of generated formulas is not empty (e.g., Aleph [45], Pratt2 [29]). This bottom formula is the most specific formula of the search space. In ILP, it is called the *bottom clause*, and is defined as the most specific clause that entails a seed example. A problem is that this bottom formula can be exponentially large, even infinite in some cases [41]. This bottom formula is also the basis for the bottom-up search. In this search, either a generalization refinement is applied to it in order to cover more positive examples; or a kind of unification operator is successively applied between 2 formulas. This latter operator is the *relative least general generalization* for first-order logic [41], the *least common subsumers* for decription logics [12], or the *longest common subsequences* for sequence patterns [4]. The first difficulty with the bottom-up search is its sensitivity to the order of consideration of examples, and to noisy data. A second difficulty is that the number of least general generalizations (*lggs*) required may be intractable, in addition to the possibly exponential size of the bottom formula.

There exists combinations of these 2 search strategies that use both specialization and generalization operators. The first method is to alter the top-down search by applying the generalization operator at some points [49]. The second approach is to start at an arbitrary point in the search space, and to apply alternatively specialization and generalization operators in order to cover more positive examples and less negative examples [16, 30]. However, the search algorithms used in bi-directional search are either greedy or stochastic, and so cannot ensure an optimal solution will be found.

This latter comment is due to the well-known trade-off between flexibility, completeness and non-redundancy of the search [5]. The more flexible the search is, the more difficult it is to make it both (weakly) complete and non-redundant. On one hand, a pure top-down search can be both weakly complete and non-redundant when its refinement operator is carefully designed [6]. But this makes it difficult to finetune the search with heuristics. On the other hand, a stochastic bidirectional search is very flexible; but it can ensure neither completeness, nor non-redundancy. One contribution of this paper is precisely to define a search strategy that is at the same time flexible, complete and non-redundant[2].

Flexibility allows the application of heuristics to guide the search towards the most promising regions of the search space. We distinguish between 2 kinds of heuristics. First, the heuristics that evaluate the proper value of a hypothesis: this enables the ranking of hypotheses as solutions to the learning problem. Second, the heuristics that evaluate the ability of a hypothesis to generate good new hypotheses (according to the first kind of heuristics): this enables the ranking of hypotheses to be refined by the search algorithm. In some cases, the same heuristic can be used in both roles. These heuristics are usually expressed as a function of the number of examples covered by the hypothesis, and also, in the case of concept learning, the number of covered positives and negatives [22]. However, according to the No Free Lunch theorems [51], it must be kept in mind that no one heuristics can be best for all applications, so that it should be specialized according to the application domain at hand.

[2]However, it must be noted that existing search stategies can very well be used along the technology of DSLs (Section 2.4), by defining the appropriate operation (specialization, generalization, least general generalizations, etc.).

3.2. Concept-Based Search

In Section 2, logical equivalence between formulas has been recognized as a cause of redundancy in search algorithms. Indeed, several syntactically different formulas can be proved to be equivalent, that is to have the same coverage whatever the dataset. These equivalences should be recognized by the search algorithm so that only one syntactical form be considered.

There is another equivalence relation between formulas that is the weakening of the logical equivalence to a particular dataset: we call it *conceptual equivalence*. Two formulas are conceptually equivalent if they cover, or discriminate, the same set of examples. The associated equivalence classes are called *formal concepts*. This notion of concept has been formalized in the theory of Formal Concept Analysis [50, 24], and its logical generalization, Logical Concept Analysis (LCA, [17]). This first requires the definition of a *context*, which roughly corresponds to a dataset.

Definition 3.1. (context and extent)
Let $\mathcal{O}$ be a set of objects, and $\mathcal{L} = (L, \sqsubseteq)$ be a logic. The triplet $K = (\mathcal{O}, \mathcal{L}, \delta)$ is called a *context*, where $\delta \in \mathcal{O} \rightarrow L$ is a mapping from objects to their logical description. An object o is said to be *covered* by a formula f in the context K iff $\delta(o) \sqsubseteq f$. The *extent* or *coverage* of a formula $f \in L$ is then defined as the set of all objects covered by f: $ext_K(f) = \{o \in \mathcal{O} \mid \delta(o) \sqsubseteq f\}$.

For instance, $K_{car} = (\{o_1, o_2, o_3\}, \mathcal{L}_{car}, \delta_{car})$ is a context, where $\mathcal{L}_{car}$ is the sub-logic starting with the functor `Conj` in Figure 2, the 3 objects are train cars, and the mapping from objects to their logical description is described in Table 2.

object	description
o_1	`axes = 2 & FlatRoof`
o_2	`axes = 3 & JaggedRoof`
o_3	`axes = 3 & Roof`

Table 2. A small context, whose objects are trains cars.

A *formal concept* can then be defined as the association of an *extent* and an *intent*.

Definition 3.2. (formal concept)
Let $(\mathcal{O}, \mathcal{L}, \delta)$ be a context. A *concept* is a pair (Ext, Int), where $Int \subseteq \mathcal{L}$ is the *intent*, $Ext \subseteq \mathcal{O}$ is the *extent*, and such that $Ext \neq \emptyset$, and Int is the non-empty class of all formulas covering all objects in Ext and no more:

$$Ext \neq \emptyset, \quad Int = \{f \in \mathcal{L} \mid ext_K(f) = Ext\}, \quad Int \neq \emptyset.$$

Not every set of objects is necessarily an extent, depending on the expressive power of the logic and the context. For instance, in the context above, the set of objects $\{o_3\}$ is not an extent, because the description of o_3 covers o_2, so that any formula covering o_3 also covers o_2; however, both $\{o_2, o_3\}$ and $\{o_2\}$ are extents. Figure 3 is a diagram of all formulas in $\mathcal{L}_{car}$ covering at least one object. These formulas are partitioned in concepts as defined above. It can be observed that there are here only 4 concepts, whereas 7 non-empty subsets of objects are possible.

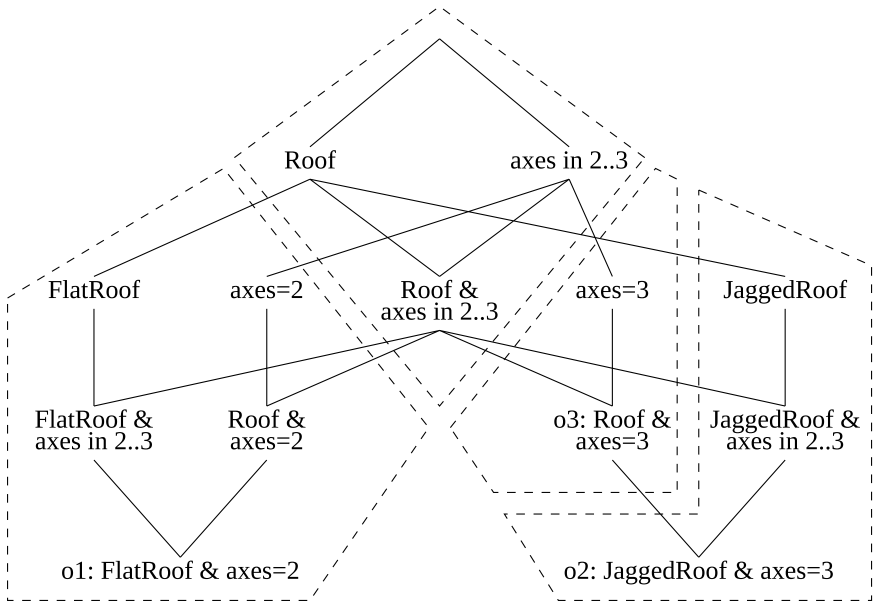

Figure 3. Logical search space partitioned into concepts.

So, a concept in the sense of "concept learning"—we call it a *target concept* to avoid confusion—is not necessarily a formal concept, for instance when it can not be discriminated in the chosen logic. However, in both kind of concepts, the focus is on the set of objects. In the target concept, no logical representation is known *a priori*; and in the formal concept, there may be many different representants. When a target concept is not a formal concept, the learning problem is over-constrained, and it can only be approximated. When a target concept is a formal concept, whose intent contains more than 1 formula, the learning problem is under-constrained, and a problem is then to choose one.

In fact, there is no easy way of choosing a concept representation in an intent. One could choose the most general representation, but such hypotheses may overgeneralize and produce false positives. The opposite approach would be to choose the most specific representants[3], but these hypotheses may overfit the training data and produce false negatives. The former choice could be called "courageous" in opposition to the latter choice that would be called "cautious" [20, 23]. Note that these 2 sets of hypotheses form respectively the upper and lower bounds of the version space. Another choice is to invoke Occam's razor and to pick the simplest hypotheses, but it has been shown that this gives no guarantee for a better generalization [15]. Because of these difficulties, we concentrate in this paper on the efficient discrimination of target concepts in expressive logics. We do not care so much about which representant of a concept is found. Anyway, if a particular representant is required, it could be searched in a post-processing step by starting from the found representant while staying in the narrow region of its concept intent.

To summarize, the logical search space is partitioned into formal concepts, and the objective is to

[3]There is an interesting connection with the maximal motifs of TEIRESIAS [43]: the maximal motifs are the most specific representants of concepts.

find a logical representant of the formal concept that discriminates well some target concept. So, ideally, we would like to search among concepts rather than formulas. Indeed, whereas the number of formulas can be infinite, the number of concepts is necessarily finite and bounded by 2^n, where n is the number of examples. However, the concepts cannot be searched directly as there is no efficient way to find a representant of a concept (otherwise, learning would be a trivial problem). The problem is therefore to design a search algorithm that takes place in the logical search space, but that approximates a search among concepts. The purpose of next section is to introduce the principles of such a search.

3.3. Dichotomic Search

As concepts can not be explored directly, but only through logical formulas, the idea is to maximize the number of visited concepts compared to the number of generated formulas. As concepts are determined by the context, this kind of exploration has to be strongly data-driven.

First, we give a better understanding of the struture of a concept intent by proving that such an intent is made of undisrupted subsumption chains.

Theorem 3.1. (intent structure)
Let x, y be 2 formulas belonging to a same concept intent Int (i.e., $ext(x) = ext(y)$). Then every formula z, such that $x \sqsubseteq z \sqsubseteq y$, also belongs to the intent Int.

Proof:
It comes directly from Definition 3.1 that, if $x \sqsubseteq z \sqsubseteq y$, then $ext(x) \subseteq ext(z) \subseteq ext(y)$. Because x and y belong to the same intent, they have the same extent. Then, z must also have the same extent, and so, belongs to the same concept intent. □

This implies that a concept is concentrated in a narrow region of the search space. This also implies that when a search progressing step by step (as most versions of top-down and bottom-up strategies) reaches a concept, it has to redundantly follow one or several chains inside the concept, except if the concept is so good or bad that the search stops or backtracks. This is illustrated in Figure 3 where every concept contains such chains of length 2 or 3.

In order to approximate a search among concepts, larger leaps should be considered rather than minimal steps. Many stochastic searches perform this kind of leap, but they are usually more syntax-driven than data-driven. Moreover, they give no guarantee w.r.t. completeness and non-redundancy of the search.

An analogy can be made from a simpler problem: the search for an element in an ordered array. The naive way is to look at the elements in the order they appear, starting from the first element. However, a much better solution is to jump into the middle of the array, and decide by a simple comparison on which side is the searched element. Half the search space is then cut off, and the procedure applies recursively on the remainder. This is called *dichotomic search*, and we adapt it to the search for hypothesis or features. The principle is to perform big leaps in the logical search space in order to reach different concepts. A difficulty is that interesting concepts can be missed by such leaps, making the search incomplete. As for the dichotomic search in an array, it is necessary to perform leaps in both directions: here, upward and downward. This requires care in the search strategy in order to avoid cycling, incompleteness, and redundancy. Section 4 develops such a strategy, proves good properties about it, and sketches an algorithm along with pruning techniques and heuristics.

4. A Generic and Anytime Search Algorithm

We develop in this section a search algorithm for learning in logics. We first detail the dichotomic search, and prove its good properties like non-redundancy and completeness. Secondly, we sketch our generic algorithm and describe where flexibility is possible and heuristics can be applied. We then show how target concepts can be used to prune and guide the search by appropriate heuristics. Our algorithm is also any-time for a better trade-off between computation time and completeness of the search.

4.1. Dichotomic Places and Operator

From the analogy of dichotomic search presented in Section 3.3, two notions are important: the *dichotomic place* and the *dichotomic operator*. In an ordered array, they are respectively an interval of positions in the array, and the localization of the middle element in this interval. In a logical search space, they need to be redefined.

A *dichotomic place* is redefined as a kind of interval of formulas $[y, Z[$, where the formula y is the lower bound, and the set of formulas Z is the upper bound. Applied to such an interval, the *dichotomic operator* must generate a set of new formulas X (equivalent to the middle element) such that for every generated formula x: (1) x is more general than y, i.e. $y \sqsubseteq x$, and (2) x is not more general than any formula in Z, i.e. $Z \not\sqsubseteq x$ (see Definition 2.1). These are added to the set F of all formulas generated so far. An additional object o is used by the dichotomic operator to ensure that every generated formula x covers more examples than y, and so reaches a different concept. This implies that o should not belong to the extent of y. In order to avoid generating the same formula twice, the set of formulas Z in a place (y, o, Z) is defined as the most specific extracted formulas that are more general than both the formula y and the object description $\delta(o)$: i.e., the least upper bounds of y and $\delta(o)$ among the generated formulas F, denoted by $lubs_F(y, \delta(o))$.

Definition 4.1. (dichotomic place)
Let $K = (\mathcal{O}, (L, \sqsubseteq), \delta)$ be a context. Let $F \subseteq L$ be the set of motifs generated so far (initially $\delta(\mathcal{O})$). A *dichotomic place* w.r.t. F is a triple (y, o, Z), where $y \in F$, $o \in \mathcal{O} \setminus ext_K(y)$, and $Z \subseteq F$ is defined as the least upper bounds of y and $\delta(o)$ in F, i.e. $lubs_F(y, \delta(o))$. It can be shortened as the pair (y, o) because Z can be deduced from it and F.

A *logical interval* is defined as an abstraction of a dichotomic place obtained by removing any reference to a context. A *dichotomic operator* is then defined as a logical operation that applies to logical intervals.

Definition 4.2. (logical interval and dichotomic operator)
Let $(L, \sqsubseteq)$ be a logic. A *logical interval* is a triplet (y, d, Z), where $y, d \in L$, $Z \subseteq L$, $y \sqsubseteq Z$, and $d \sqsubseteq Z$. A *dichotomic operator* is a function from any logical interval (y, d, Z) to a set of formulas $X \subseteq L$ such that for all $x \in X$: (1) $y \sqsubseteq x$, (2) $d \sqsubseteq x$, and (3) $Z \not\sqsubseteq x$.

According to the above definitions, a new formula x is produced as a generalization of an already generated formula y and an object o not covered by y. In these terms, the dichotomic search is similar to bottom-up strategies. But there is a major difference: the generated formula does not need to be a least general generalization, and the set of generated formulas does not need to be all the least general generalizations (*lggs*). Typically in searching there is a trade-off between using an expressive logic and having very specifi c and numerous *lggs*, or using a simpler logic and risking missing the right formula. In dichotomic search this trade-off evaporates, as it is possible to fi rst generate a limited set of prefered generalizations, and then refi ne them by further applications of the dichotomic operator if necessary.

The idea of applying the dichotomic operator several times to the same place raises the question of redundancy in the generation of formulas. In other words, what makes the application of the dichotomic operator to the same dichotomic place generate different formulas ? We demonstrate 3 results showing the good properties of dichotomic search w.r.t. redundancy.

Theorem 4.1. Every generated formula covers at least 2 examples of the context.

Proof:
According to Defi nition 4.2, every formula x generated from a place (y, o, Z) is more general than both y and $\delta(o)$. This entails that if y covers at least one example, then x covers at least 2 examples. As initially, the y's are object descriptions (Defi nition 4.1), which cover at least one example each, this terminates the proof. □

This means that dichotomic search is strongly data-driven, and thus avoids consideration of formulas that are irrelevant to the context. This compares well with common top-down search, in which formulas covering no object are considered (even if this is a termination condition for the search). The use of a bottom formula can ensure that generated formulas cover at least 1 example, but not 2. Even bottom-up strategies can not ensure this result, when they apply generalization steps to the bottom clause instead of using *lggs*.

Theorem 4.2. Let F be the set of formulas generated so far. Every generated feature x is new w.r.t. the set F of generated formulas: i.e., $x \notin F$.

Proof:
Suppose x is a feature generated from some dichotomic place (y, o, Z), and already belongs to F. Then from Defi nition 4.2, we have $y \sqsubseteq x$, $\delta(o) \sqsubseteq x$, $Z \not\sqsubseteq x$, and we also have $x \in F$. This entails that $Z = lubs_F(y, \delta(o)) \sqsubseteq x$, which is contradictory. □

This result is very important as it proves that, while the search is very fexible (no order is specifi ed for the visit to dichotomic places) and bidirectional (the generated formulas can be used both in lower and upper logical bounds), there is no redundancy in the search. In particular this eliminates the danger of cycling. So, what happens when the dichotomic operator is applied several times to the same dichotomic place ? First, a dichotomic place is characterized by a pair (y, o), the upper bound Z of the logical interval being deducible as $lubs_F(y, \delta(o))$. When a formula x is generated from this place, it is inserted into F, which results in the update of Z as $lubs_{F\cup\{x\}}(y, \delta(o))$. From Defi nition 4.2, it entails that x belongs to the new defi nition of Z: this prevents it being regenerated from (y, o). So, Z acts as a memory of what has already been generated from the place (y, o), or any other places. Further generated formulas are generated either between (y, o) and $lubs_{F\cup\{x\}}(y, \delta(o))$, or between (x, o') and $lubs_{F\cup\{x\}}(x, \delta(o'))$ where o' is an object not covered by x. Hence the idea of dichotomy. A difference here with the dichotomic search in an array is that the search has to proceed in both halves of the logical interval.

Theorem 4.3. Let F the set of formulas generated so far. For every formula x generated from a place (y, o, Z), the concept of x is different from the concept of y or $\delta(o)$.

Proof:
As x is a generalization of y and $\delta(o)$ (Definition 4.2), we have $ext(y) \cup ext(\delta(o)) \subseteq ext(x)$. Moreover, $ext(y) \neq \emptyset$ (Theorem 4.1) and $o \notin ext(y)$. This entails that $ext(x)$ is strictly bigger than both $ext(y)$ and $ext(\delta(o))$, and so belongs to a different concept (Definition 3.1). □

Theorem 4.3 supports the idea of exploring as many concepts as possible by forcing a generated formula x to reach a different concept than the concept of the lower bound y. This is made possible by the use of an additional object. It is tempting to make this concept also different from the concept of any upper bound $z \in Z$. A solution would be to use another object o' covered by all upper bounds in Z but not covered by the lower bound y. A generated formula would have to cover o but not o'. However it may be difficult to handle this negative constraint in the definition of dichotomic operators. More importantly this would make the dichotomic search incomplete because, even if a generated formula belongs to the concept of an upper bound, it may be useful as a lower bound to generate more general formulas and reach new concepts. We now prove the completeness of the dichotomic search, under the assumption that the dichotomic operator is itself complete.

First, given a dichotomic operator $dicho$, we define its recursive application on a logical interval as the union of the results of its successive applications.

Definition 4.3. (recursive dichotomic operator)
Let (y, d, Z) be a logical interval. The nth application of the dichotomic operator, denoted by $dicho_n(y, d, Z)$, is defined by:

$$\begin{cases} dicho_0(y,d,Z) & = & \emptyset \\ dicho_{n+1}(y,d,Z) & = & dicho_n(y,d,Z) \cup dicho(y,d,Z \cup dicho_n(y,d,Z)). \end{cases}$$

This definition helps us to define the completeness of a dichotomic operator as its ability to approximate every formula that subsumes the 2 lower bounds of a logical interval.

Definition 4.4. (complete dichotomic operator)
A dichotomic operator $dicho$ is *complete* iff for every logical interval (y, d, Z), and every formula x s.t. $y \sqsubseteq x$, $d \sqsubseteq x$, $Z \not\sqsubseteq x$, there exists a finite number n and a formula $x' \in dicho_n(y, d, Z)$ such that $x' \sqsubseteq x$.

The reason for using the weaker condition $x' \sqsubseteq x$ instead of $x' \equiv x$ is that this would ruin the very idea of dichotomy because this would suppose a step by step exploration. Indeed, the role of the dichotomic operator is to find formulas somewhere between the lower bound y and the upper bound Z. If x' is too specific, it can be generalized by the application of the dichotomic operator to another place (x', o', Z'). Furthermore, if they represent the same concept, x' can be used instead of x as the objective is to discriminate concepts.

We now prove in the following the important result that dichotomic search is complete w.r.t. the discrimination of concepts.

Theorem 4.4. Let $dicho$ be a complete dichotomic operator, and F be the set of formulas generated so far. For every formula $x \notin F$ such that $ext(x) \neq \emptyset$, and for which there is no more specific concept representant in F (i.e., $\nexists x' \in F : x' \sqsubseteq x, ext(x') = ext(x)$), a formula x' more specific than x and belonging to the same concept can be generated by a finite number of dichotomic operations.

Proof:
Firstly, we identify a dichotomic place (y, o, Z) from which x' is generated. Let $y \in F$ be a formula such that $y \sqsubseteq x$. It must exist because $ext(x)$ is not empty and so subsumes at least one object description, and F initially contains all object descriptions. Now, if $ext(y) = ext(x)$, this contradicts the hypothesis that x has no more specifi c concept representant. Otherwise, there must be an object o that is covered by x and not by y. Then, (y, o, Z) is a dichotomic place, where $Z = lubs_F(y, \delta(o))$. If there exists an upper bound $z \in Z$ such that $z \sqsubseteq x$, then we start again by using z instead of y as a lower bound. This process of replacement of the lower bound will terminate because the extent of z is strictly bigger than the extent of y (addition of the object o), and the extent of x is fi nite. Otherwise, x is an acceptable generated feature for the dichotomic place (y, o, Z).

Secondly, from Defi nition 4.4, we deduce that a formula $x' \sqsubseteq x$ can be generated by a fi nite number of application of the dichotomic operator to the logical interval $(y, \delta(o), Z)$. If $ext(x') = ext(x)$, then the conclusion is proved. Otherwise, we start again at the beginning of this proof by replacing y by the new formula x'. This process of replacement will terminate because we also know from Defi nition 4.2 that x' has a bigger extent than y because it covers both $ext_K(y)$ and o. □

This theorem can be given a simpler formulation by concentrating on the discrimination of concepts.

Corollary 4.1. Let K be a context. By a dichotomic search starting with $F = \delta(\mathcal{O})$, every concept of K can be discriminated, i.e. can be found a logical representant, in a fi nite number of operations.

Proof:
Let c be a concept for which there is no representant in F, and $x \notin F$ be an unknown representant. From Defi nition 3.2, $ext(x) \neq \emptyset$ because concepts have non-empty extents. According to Theorem 4.4, a representant $x' \sqsubseteq x$ of the concept c can be generated in a fi nite number of operations. □

The proof of Theorem 4.4 exhibits 3 iterations. A fi rst iteration through the dichotomic places in order to fi nd the dichotomic place that can generate the formula x. A second iteration as the recursive application of the dichotomic operator, until a specialization x' of x is found. A third iteration to further generalize x' until reaching the concept of x; this implies to build new dichotomic places from the newly generated formulas.

In practice, of course, the search cannot be directed towards a unique formula x. Indeed these iterations can be freely mixed up, giving great fl exibility to the search. However, depending on the logic used, there may be some constraints on the order of visiting places. To each dichotomic place we can associate a process that consists of repeatedly applying the dichotomic operator to this place, updating the upper bound of the place, and creating new processes for the new generated formulas. A process terminates when the dichotomic operator cannot generate formulas any more. If all processes terminate, then execution resources can be allocated in any order without breaking completeness. On the contrary, the allocation of execution resources must be fair in order to keep completeness. In fact, such a situation can occur only when the *lggs* of 2 formulas are not defi ned in the logic, which means there are downward infi nite chains in the upper bounds of the 2 formulas. This is certainly not a common situation, but the ability to learn in such logics could appear to be useful in some continuous domains.

There is a symetrical situation where the minimal specializations of a formula are not defi ned because there are upward infi nite chains in the logic. This is more common than with downward infi nite chains. For instance, there are no minimal refi nements of the interval $]-\infty, +\infty[$; and they are no minimal refinements of the regular expression $a*$ either if we accept expressions like $a\{3,6\}$. To the best of our knowledge, our dichotomic search is the only one that allows for a complete search in a search space that contains both upward and downward infinite chains.

Theorem 4.4 (and its corollary) is a strong result as it proves that concept discrimination, i.e. machine learning, can be achieved in a finite amount of time, even if the logical search space is infinite, and even if the *lggs* are undefined or infinite. This result cannot be achieved by top-down and bottom-up search without restricting the domain of applicable logics.

4.2. Algorithm and Complexity

A pseudo-algorithm is presented below. Given a context $K = (\mathcal{O}, (L, \sqsubseteq), \delta)$, it starts with an initial set of motifs F, and an initial set of dichotomic places P derived from F. In fact, only the 2 first components (y, o) of dichotomic places are stored in P. Indeed the third component $Z = lubs_F(y, \delta(o))$ depends on the current set of features F, which may grow between the time the place is created, and the time the dichotomic operator is applied on it. These places are used to generate new motifs by the dichotomic operator, which produce new places. A place is removed when it no longer generates motifs, and the whole process terminates when no place is remaining.

```
01. $F \leftarrow \delta(\mathcal{O})$;
02. $P \leftarrow \{(y,o) \in F \times \mathcal{O} \mid o \notin ext_K(y)\}$;
03. while $P \neq \emptyset$ do
04.     $(y,o) \leftarrow choice(P)$;
05.     $Z \leftarrow lubs_F(y, \delta(o))$;
06.     $X \leftarrow dicho(y, \delta(o), Z)$;
07.     if $X = \emptyset$ then
08.         $P \leftarrow P \setminus \{(y,o)\}$;
09.     else
10.         $F \leftarrow F \cup X$;
11.         $P \leftarrow P \cup \{(x,o) \in X \times \mathcal{O} | o \notin ext_K(x)\}$;
12. done.
```

Naturally, such an exhaustive search may not be tractable, because the number of objects or the search space is too large w.r.t. available resources. When a search algorithm has not terminated after a long time, it is often preferable to acquire the results found so far, rather than merely killing the process. After obtaining such intermediate result the process may then be left running so that more results can be found. An algorithm that can give results whenever required is said to be *any-time*. In our implementation, the search runs in a background process, and its results can be displayed and processed by a foreground process at any time. Of course, most search algorithms could easily be made any-time, however we consider this most appropriate to dichotomic search. For example, top-down search explores the search space using only small steps, in order to avoid redundancies in search, and because it progresses only in one direction. This means that given a small proportion of the termination time, the algorithm will have explored only a small proportion of the search space. Therefore, a partial result given by an any-time algorithm would tend to be strongly biased. In contrast a dichotomic search performs large leaps, and quickly reaches large parts of the search space. Moreover, the order in which dichotomic places are considered is left completely free, which helps to concentrate the generation of interesting motifs at the beginning of the search.

The output of the search algorithm is a set of formulas representing all possible concepts of the context K. So, this is enough for the feature mining task: a propositional learner can be applied to the set F of formulas as a set of features in order to produce rules and/or hypotheses as combinations of features (see Section 2.6). For the tasks of rule and hypothesis discovery, it is sufficient in theory to select the best elements of F according to the target concept to be learned. In practice, and in order to keep the algorithm any-time, it is preferable to add a variable in the algorithm for registring the best rules/hypotheses at any time of the search.

In our implementation, the set F of generated formulas is internally represented as the Hasse diagram of the subsumption ordering $(F, \sqsubseteq)$. This representation has the following advantages compared to a list, for instance:

- the potentially costly subsumption test is used only for inserting a new element in F, and nowhere else in the algorithm,
- the extent of a formula can then be computed by a simple downward traversal of the diagram, what is in fact done and cached at the formula insertion,
- the *lubs* of 2 elements can also be computed by traversals of the diagram, upwards this time, which saves many subsumption tests.

The flexibility of our algorithm resides in the fact that at line 04 it is possible to choose any place in P on which to apply the dichotomic operator. Thus, it is natural to associate a heuristic value $h(y, o)$ to every place $(y, o) \in P$, and to choose each time the best place according to this heuristic. This helps to concentrate the generation of interesting features/rules/hypotheses towards the beginning of the search, as required by an any-time approach. This heuristic is discussed in Section 4.3. As this heuristic entails an ordering of places, we internally represent P by a binary tree. This representation enables the choice of the best place and the addition of new places in a time $O(log|P|)$, instead of $O(|P|)$ with a list.

We now discuss the complexity of our algorithm. Firstly, we define $n = |\mathcal{O}|$ as the number of objects in the context, and $N = |F|$ as the number of generated formulas (including the n object descriptions). Secondly, the number of places $|P|$ is bounded by Nn. Thirdly, we assume the dichotomic operator generates only one formula at a time. This is always possible because of the available flexibility in its definition; and this can only increase the worst-case complexity. The complexity of each significant line in the above algorithm is as follows:

- 04. $O(log(Nn))$: retrieval of the best place,
- 05. $O(N)$: computation of the $lubs$,
- 06. $O(|dicho|)$: application of the dichotomic operator,
- 10. $O(N|\sqsubseteq|)$: insertion of the new formula in the Hasse diagram $(F, \sqsubseteq)$,
- 11. $O(n(|h| + log(Nn)))$: computation of the new places, their heuristic value, and their insertion in P.

By integrating these local complexities over N generated formulas, we find that the global worst-case complexity of the algorithm is $O(N^2| \sqsubseteq | + N|dicho| + Nn(|h| + log(N)))$. The first term is the most important and reflects the insertion of new formulas in the subsumption diagram (line 10). The second term corresponds to the application of the dichotomic operator (line 06), and is important given that $dicho$ can have a high complexity. The third term expresses the cost of evaluating and sorting places in P (line 11). Various experiments have confirmed that the 2 major costs are the insertion of new formulas in the diagram and the dichotomic operator. They also agree with the theoretical complexity in that the first term takes an increasing part of the running time when more and more formulas are generated.

4.3. Supervised Learning

Usually, one is not interested in the discrimination of every possible concept, but in the discrimination of a set T of predefined target concepts (supervised learning). Each target $t \in T$ partitions the set of objects in positive examples E_t^+ and negative examples E_t^-. For example, in the Michalski's train context given in Figure 1, there are two target concepts: the East-bound trains, and the West-bound trains. In this example each object belongs to only one target (as a positive example), but this is not necessary and we consider that an object can belong to several targets. In Section 6.2 we present a biological application in which objects are proteins and can have several different functions. So, our learning setting is supervised, multi-targets, and multi-labels.

The existence of target concepts enables us to define some concepts as better than others, and to identify *optimal* concepts.

Definition 4.5. Let $K = (\mathcal{O}, \mathcal{L}, \delta)$ be a context, and T be a set of target concepts. A concept c_1 is said strictly *better* than a concept c_2 w.r.t. t, what we note $c_1 >_t c_2$, if the former either covers more positives and no more negatives, or covers less negatives and no less positives in t. A concept c is said *optimal* w.r.t. t if it is maximal for the partial ordering $<_t$. A concept c is said optimal w.r.t. the set of targets T if it is optimal for some target $t \in T$.

A concept is optimal for some target t when it is not possible to cover more positive examples without covering more negative examples, and it is not possible to cover less negative examples without covering less positive examples. In the ideal case where the target concept t is also a formal concept, then this concept is the only optimal one, and it matches exactly the target (covering all positive examples, and no negative examples).

The focus on a limited set of target concepts should make it possible to prune the search by not considering places irrelevant to them. To this purpose, we first define when a place is relevant to a set of targets, and then prove that the discrimination of optimal concepts is preserved.

Definition 4.6. Let $K = (\mathcal{O}, \mathcal{L}, \delta)$ be a context, and T a set of target concepts. A dichotomic place (y, o) is said to be *relevant* to the set of targets T iff

$$\exists t \in T : ext_K(y) \cap E_t^+ \neq \emptyset \wedge o \in E_t^+.$$

Theorem 4.5. Let K be a context, and T be a set of target concepts. By a dichotomic search starting with $F = \delta(\mathcal{O})$, every concept optimal for T can be discriminated in a finite number of operations.

Proof:
The proof of this theorem is similar to the proofs of Theorem 4.4 and Corollary 4.1, except the dichotomic place (y, o) is constrained to be relevant to some target: y contains at least one positive example, and o is a positive example. These constraints are easily satisfied by considering that the target has at least 2 positive examples. The other cases are trivial. □

According to Definition 4.5 of *better concept* and *optimal concept*, the measure of the quality of a concept w.r.t. some target is a combination of the number of positive examples and the number of negative examples it contains. This is more general than most other measures [22], as they are total orderings that can be derived from it: e.g., accuracy, entropy, Laplace estimate. A natural condition for this total ordering is to be compatible with the partial ordering "better than", which is generally the case.

A question that may arise is: which of 2 different optimal concepts is to be prefered ? If our strategy is to produce many features to be used by a propositional learner, a reasonable answer is *both* as we want to give as much information as possible to the learner. If our strategy is to produce rules that will be used for prediction, the choice depends on which kind of errors we prefer: false positives or false negatives ? As this depends on the application domain (false positives are prefered in security, while false negatives are prefered in functional genomics), we purport this choice by allowing the end-user to filter the set of optimal concepts according to her own criterion. This approach is supported by the No Free Lunch theorems [51], which imply that no measure can be good for all domains. These comments also apply to the heuristics used to select a dichotomic place (see the algorithm in Section 4.2).

4.4. Learning Methodology for Overfitting Avoidance

One of our motivation for a concept-based, dichotomic and any-time search algorithm is to allow the use of expressive logics, in order to find complex patterns when this is necessary. This approach can receive two paradoxical criticisms:

- With expressive logics the completeness of the search is intractable in practice, so that found results are not optimal;
- With expressive logics and arbitrarily long searches there is a high risk of overfitting the training data.

The only way to reconcile these two views would be to restrict ourselves to less expressive logics or to introduce some bias. But such bias may not be available because of insufficient background knowledge; and asking users to try and test numerous parameter settings only shifts the burden of the search onto them. In fact, it has been shown that the overfitting problem is not directly related to the size of the search space, but rather to the number of tests [15, 28]. In our algorithm the number of tests is equal to the number of generated formulas. It is important to note that the exploration of the space of parameter settings by the users of learning systems also count as tests.

In fact, the best solution to the above problem is to use as much background knowledge as possible [47]. Indeed, this restricts the search space in a way that does not exclude relevant solutions. However, background knowledge may be insufficient to reconcile completeness and overfitting avoidance, so that some additional strategy is necessary.

There exists several strategies for correcting the effects of multiple testing [28]. Our strategy is to use a *validation dataset*, in addition to the training dataset, and a distinct test dataset. We first use the training dataset to generate a large number of formulas (as features, rules or hypotheses). If these formulas are generated as features, a learner (e.g., C4.5 [42]) can be applied to produce rules or hypotheses. Now, the more formulas are generated, the more likely it is to find apparently interesting rules/hypotheses by chance. So, we then use the validation dataset to evaluate the statistical significance of the results of the training stage on new data. For this, we use a hypergeometric distribution with a Bonferroni correction in the case where several rules/hypotheses are evaluated. The most significant rules/hypotheses ($P \leq 0.05$) are finally applied to the test dataset for a final and unbiased evaluation. This methodology is derived from the Data Mining Prediction (DMP) framework [11], and is summarized in Figure 4. This methodology can be combined with cross-validation, and other methods are also possible.

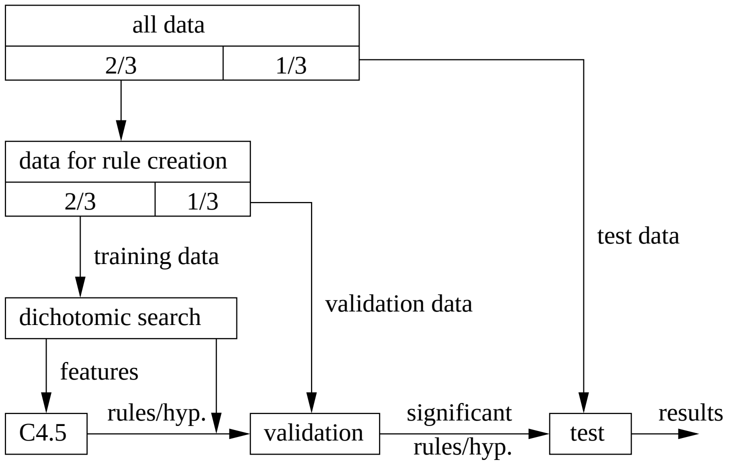

Figure 4. The Data Mining Prediction methodology.

5. Dichotomy in Logics

In Section 4 a dichotomic search algorithm is defined and discussed, but the dichotomic operator was left abstract as it depends on the chosen logic. In order to make our method well-founded we have to show that this unusual operator can effectively be defined and implemented for various concrete logics. This section aims to fill the gap by defining an algorithm for the dichotomic operator of increasingly complex logic functors (Section 2.5): intervals (`Interv`), pairs of formulas (`Pair`), and sequence motifs (`Seq`).

5.1. Intervals

Given a logic of totally ordered values `Val`, the logic `Interv(Val)` enables us to represent and reason on intervals over these values. In this logic, the subsumption relation corresponds to interval inclusion, where larger intervals are more general. The total ordering on values implies that the maximum and minimum of 2 values can be computed. Hence we propose the following algorithm for the dichotomic operator of a logic `Interv(Val)`, where the sub-logic `Val` remains abstract.

```
Interv(Val).dicho([f1,f2],[d1,d2],Z):
    x1 $\leftarrow$ Val.min(f1,d1);
    x2 $\leftarrow$ Val.max(f2,d2);
    if Z $\not\sqsubseteq$ [x1,x2]
    then X $\leftarrow$ $\{$[x1,x2]$\}$
    else X $\leftarrow$ $\emptyset$;
    return X.
```

We recall that generated features must be more general than both `f = [f1,f2]` and `d = [d1,d2]`, but not more general than any interval in `Z`. These conditions are obviously satisfied as the generated feature is the union of intervals `f` and `d`, and is checked not to be more general than `Z`. In fact, in this case, the dichotomic operator returns the *lggs* (least general generalisations) when they are not already in `Z`, and nothing otherwise. From this it follows that this operator is necessarily complete (Definition 4.4) because every acceptable generated feature is necessarily more general than the *lggs*.

This also shows that dichotomic search can mimic a bottom-up search when the *lggs* are defined and efficiently computable. This is the case of intervals where there is only one *lgg*.

5.2. Pairs

Given 2 logics `L1` and `L2`, the logic `Pair(L1,L2)` enables us to represent and reason on pairs of formulas `(f1,f2)`. This can be used to represent valued attributes or vectors. A formula `(f1,f2)` subsumes a formula `(g1,g2)` iff `f1` (resp. `f2`) subsumes `g1` (resp. `g2`). The principle of the dichotomic operator is to apply the dichotomic operator of each sub-logic to its respective sub-formulas, and then to build new pairs from the generated sub-formulas. In the case where no sub-formula is generated on any side, sub-formulas in `Z` have to be used instead to avoid incompleteness of the dichotomic operator.

```
Pair(L1,L2).dicho((f1,f2),(d1,d2),Z):
    Z1 $\leftarrow$ $\{$z1 $|$ (z1,.) $\in$ Z$\}$;
    Z2 $\leftarrow$ $\{$z2 $|$ (.,z2) $\in$ Z$\}$;
    X1 $\leftarrow$ L1.dicho(f1,d1,Z1);
    if X1 = $\emptyset$ then X1 $\leftarrow$ Z1;
    X2 $\leftarrow$ L2.dicho(f2,d2,Z2);
    if X2 = $\emptyset$ then X2 $\leftarrow$ Z2;
    X $\leftarrow$ (X1 $\times$ X2) $\setminus$ Z;
    return X.
```

The correctness of generated formulas `(x1,x2)` can easily be checked by considering the 4 cases depending on whether `x1` $\in$ `Z1` and whether `x2` $\in$ `Z2`. The completeness of the dichotomic operator is also satisfied provided that the dichotomic operators of sub-logics are also complete.

5.3. Sequences

Given `Elt`, a logic of elements, the logic `Seq(Elt)` enables us to represent sequences and motifs over these elements. As shown in Section 2.5, motifs are made of elements and flexible gaps, whose length is defined by an interval (minimum length and maximal length). The use of a logic for elements allows them to be more complex than mere alphabetical symbols. To fi x the ideas, we consider the representation of the secondary structure of proteins, where elements can be represented by valued attributes: there are 3 attributes corresponding to three types of structural elements (helix, sheet, coiled coil) and respectively denoted by `a,b,c`; the values are integer intervals corresponding to the length of each element in number of amino-acids. So, these elements can be represented in the logic `Pair(Attr(a,b,c),Interv(Int))`.

The algorithm of the dichotomic operator is adapted from the computation of LCS (Longest Common Subsequences) [4, 7]. This adaptation is necessary because we consider complex elements and gaps described by length intervals. The fi rst step is the computation of an alignment matrix that is explained by the example in Figure 5. In this example, the motif at the top and the sequence at the left are respectively the formulas y and d of a logical interval (y, d, Z). At each intersection of an element line and element column, the dichotomic operator of the sub-logic `Elt` is applied to these 2 elements and subsuming elements in Z in order to produce consensus elements. These consensus elements are linked according to the LCS algorithm to represent gaps, and described by a length interval. Numerical constraints can be applied to fi lter out elements and gaps: in the example, element length is not allowed to vary in a ratio greater than 3, and gaps must be no longer than 2 elements.

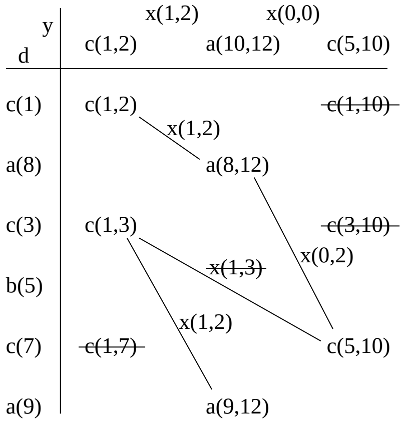

Figure 5. Generation of 2 motifs generalizing a motif and a sequence.

The second step is to extract a graph from the matrix, whose paths defi ne motifs subsuming both y and d. The fi nal step consists in transforming this graph in a set of longest motifs. In the example of Figure 5, these motifs are `c(1,2)-x(1,2)-a(8,10)-x(0,2)-c(5,10)` and `c(1,3)-x(1,2)-a(9,10)`. They can fi nally be fi ltered by removing those that are more general than Z.

If the dichotomic operator of the sub-logic of elements generates *lggs* (which is the case in the above example), then the above algorithm also generates *lggs* for sequences and motifs. As the number of *lggs* can be very large, it is a good idea to try and make this operator as dichotomic as possible. A solution is to consider increasing values for numerical constraints. For example, gap-less motifs can be generated fi rst, then motifs with gaps of maximum length 1 element, and so on until reaching the user-specifi ed bound. This is reminiscent of language hierarchies and bias shift [52], but a major difference is that this applies locally to each dichotomic place. This means it is not necessary to exhaust the search at one level before exploring further levels. The complete algorithm of the dichotomic operator for sequences and motifs is now summarized below.

```
Seq(Elt).dicho(f,d,Z):
M $\leftarrow$ alignment-matrix(f,d,Z,max-gap);
G $\leftarrow$ motifs-graph(M);
X $\leftarrow$ $\emptyset$; i $\leftarrow$ 0;
while X = $\emptyset$ and i $\leq$ max-gap do
   X $\leftarrow$ longest-motifs(G,Z,i);
   i $\leftarrow$ i+1;
done;
return X.
```

6. Examples and Applications

We demonstrate the applicability of our learning method with 2 examples: (1) the East-West train problem, and (2) a bioinformatics application with the prediction of protein functions from their secondary structure. These experimentations are based on a prototype of dichotomic search algorithm, which is implemented in Objective Caml[4]. One of the advantages of this programming language is to provide the technology for implementing logic functors. The chosen measure for selecting the dichotomic place is maximum confidence w.r.t. all targets, i.e. the number of true positives out of the number of covered examples.

6.1. The East-West Train Problem

In a first experiment, we consider the context made of the 10 original trains (see Figure 1, page 1003), which are described according to Section 2.5. The target concept is the set of 5 East trains, and the allowed patterns are as described in Section 2.5. As this is a small problem, we were able to let the search run until termination. This took a total of 199.56s, and resulted in the generation of 875 formulas. It can be noted that this number is close to the maximum number of concepts containing at least 2 positives examples (as this is guaranteed by the dichotomic search): 832 concepts. The total computation time is decomposed in 160s for the dichotomic operator, 30s for the insertion of new formulas in the Hasse diagram, and 10s for the remaining.

Among the generated formulas, 16 discriminate perfectly the target concept, which means they are all representations of the same concept; in fact hypotheses of the target concept. The first one was generated after only 0.94s, and the last after 108.26s. So, the search could have been stopped after only 0.94s, but it is worth letting the search continue as more compact/interesting/interpretable representants of the target concept may be found later. The shortest representant, which is also the first generated, is `train match loads = 1 - LTriangle`: "a train is East-bound iff it contains a car that has only one load and is followed by a car containing one or several triangle loads". A more general version of it is also generated: `train match loads in 1..2 - x(0,1) - LTriangle`. Compared to the more specific version, this says that the first car can also contain 2 loads, and that there can be an optional car between the first and second car of the first pattern. The well-known hypothesis `train match Roof & Short` is also found after 2.21s, but in a specialized version: `train match NotDouble & Roof & Short`

[4]`http://pauillac.inria.fr/caml`.

`- axes in 2..3 & loads in 1..3`. The more general version can easily be obtained from it by removing redundant parts.

In a second experiment, we considered the East-West train challenge [34] that adds 10 new trains to the 10 existing trains. This time we stopped the search after a time similar to the first experiment (243s), which resulted in the generation of 1264 formulas. No representant of the target concept was found this time, and the best found formulas cover 8 positive examples (out of 10), and no negative examples. This makes them good rules, but not good enough hypotheses. So, we applied a propositional learner (here, the decision tree learner C4.5) in order to form a set of rules covering all positive examples. The result is a hypothesis made of the 2 following rules:

```
east ←
   train match NotDouble - loads = 1 - LTriangle
east ←
   train match loads in 1..2 - Short - NotDouble & loads = 1 & axes = 2
```

The first rule covers 8 objects and has been generated after 1.62s, while the second rule covers 7 objects and has been generated after 188.75s. This is consistent with known hypotheses, which are all made of 2 rules. We may obtain shorter rules by postprocessing the generated formulas: each formula would be reduced by removing redundant parts, and only shortest representants of each concept would be kept and passed to the propositional learner. Here, we have chosen not to apply any post-processing step in order to give a clearer illustration of our algorithm.

6.2. Predicting Protein Functions from Secondary Structure

We also applied our learning approach to the problem of predicting a protein's function from its sequence. In bioinformatics the key source of information about proteins is their sequence. In theory a protein's sequence contains all the necessary information to determine its 3D structure, which in turn determines its biochemical functions. Unfortunately, the "protein folding problem", the problem of determining the 3D structure of proteins from its sequence, is still unsolved [48]. It is also an unsolved problem to predict a protein's function from its 3D structure. Our approach is to first predict a protein's secondary structure (its local 3D structure), which is known to be possible at a reasonably good accuracy (e.g., by PROF [39]), and then to learn predictive patterns for function in secondary structure.

Protein secondary structure is a sequence of structural elements like helices, sheets, and coiled coils. This makes it possible to use a logic similar to the logic used in the train example (Section 2.5), just changing the type of the sequence elements (see Section 5.3), and the attribute name `train` into `struc`. The resulting logic is defined by the following combination of logic functors:

```
L = Prop(AIK(Pair(Attr(struc),Seq(Pair(Attr(a,b,c),Interv(Int)))))).
```

In our experiments we considered the whole genome of Yeast (*S.cerevisiae*) [26], in which about 30% of the ∼6000 proteins have still no known function – despite it being one of the most studied organisms. We applied the DMP methodology presented in Section 4.4 and split the 3827 proteins with known functions in 3 datasets: a training dataset (1660 example), a validation dataset (855 example), and a test dataset (1312 examples). The target concepts are taken from the MIPS[5] hierarchy of function

[5]`http://mips.gsf.de/proj/yeast/CYGD`.

classes. We used the 3 first levels of the hierarchy, i.e. 169 target concepts. Each protein can belong to several function classes, and often does.

Each target concept was learned separately in an average of 85min of computation on an AMD Athlon 1.66GHz 1Gb. The total computation time required for the 169 classes was 240h, distributed over a cluster of 30 identical machines. Each search process was stopped when the memory used reached 900Mb. The number of generated motifs is $\sim$10,000 for most classes, and 30,000 at most. The decision tree learner C4.5 was then applied to these motifs to produce classification rules, which were filtered on the validation dataset and evaluated on the test dataset (see Section 4.4 for more details[6]).

Significant rules were found for 33 classes, and 20 of them gave significant results in the test dataset ($P < 0.05$). This may seem disappointing, however the task is known to be a very hard one. In a previous experiment [11] using the same datasets and more computation time, Warmr [13] was used instead of our algorithm to generate features, and significant rules were found for only 6 classes instead of 33. One important difference lies in the expressivity of the motif language. In the previous experiment, motifs contain at most 3 elements, no gaps, and the length of structural elements are discretized in 5 values, whereas we considered motifs of arbitrary length with flexible gaps and free intervals over the length of elements (see Section 5.3).

Among the classes with significant rules, 7 have very significant results with $P < 10^{-5}$. The striking fact is that these classes are all transport and transporter classes, whose proteins are located in the membrane. This implies two biologically surprising conclusions: (1) the secondary structure predictions made by PROF works well with membrane proteins, while PROF was trained only on globular proteins, and (2) the relationship between secondary structure and function is stronger in membrane proteins than in globular proteins.

```
'C-compound, carbohydrate transport' ¡-
        struc match (c(3,8)-x(0,2)-b(3,6)-a(10,18)-x(0,1)-c(5,15)-
            x(0,2)-a(7,15)-x(0,2)-c(5,10)-x(0,2)-a(3,6)-c(2,4))
```

proteins: YDR342c, YDR387c, YDR497c, YHR094c, YHR096c, YJR158w, YOL156w, YOL103w
precision = 100% (8/8), recall = 53% (8/15), $P = 10^{-16.5}$

Figure 6. Example of a rule discovered for the class "C-compound, carbohydrate transport".

As an example, a rule discovered for the class "C-compound, carbohydrate transport" is given in Figure 6, along with the 8 ORFs it covers in the test dataset. They all belong to the class so that the precision of the rule is 100%. As this rule also covers more than half the class (recall = 53%), it is highly statistically significant ($P = 10^{-16.5}$). The rule body is made of one motif. This motif exhibits a sequence of 3 helices (i.e., `a(10,18)`, `a(7,15)`, `a(3,6)`), 2 of medium length and one short, separated by medium length coiled coils plus possibly additional elements. This is consistent with the knowledge that membrane proteins contain several helices as transmembrane regions.

The application of the significant rules to the proteins with no known function yields 19 predictions with confidence >50%. The prediction that the protein YPR128c may have the function "Mitochondrial transport" has already been confirmed in MIPS.

[6]Complete results are available at `http://www.aber.ac.uk/compsci/Research/bio/dss/yeast.ss.mips/`.

7. Discussion

There are still several issues that need to be clarified in order to get a better understanding, and better applications, of dichotomic search and DSLs. The most prominent issue is probably that of exploiting the available flexibility. The idea here is to guide the search towards the best elements by making use of as much of the available data and background knowledge as possible. To this end more work is needed in the definition of logics; especially in the dichotomic operator, and search heuristics. The fact that our algorithm is any-time should also make it possible to reconcile different search strategies: for example, a search that behaves like a separate-and-conquer search at the beginning, but that is still complete in the limit. More work is also necessary about the best use of generated formulas: (1) how to post-process them given that they are arbitrary representants of concepts, and (2) which propositional learner is best suited to them ? Finally, as the application of the dichotomic operator is independent from one dichotomic place to another, it should be possible to parallelize it by distributing places on different nodes, and only sharing the set of generated formulas.

8. Conclusion

We have introduced a new concept-based and dichotomic search learning algorithm that is generic w.r.t. to the logic used to represent examples, features, rules, and hypotheses. The first advantage of this is that instead of having the logic fixed as propositional or predicate logic, a wide range of intermediate logics can be used. This enables adjustment of the representation language to the application domain, hence the name of *domain-specific logics* (DSL). These DSLs can be assembled from logic components, the *logic functors*, which correspond to different data structures. The second advantage is that learning can be applied to more types of logic (both upward and downward infinite chains are allowed), to search more complex patterns, and in more efficient ways, thanks to specialized implementations of logical operations (subsumption, *dichotomic operator*). A third advantage is to provide more flexibility in the search as *dichotomic places* can be visited in any order, while retaining both completeness and non-redundancy in the search. There is also some flexibility in the dichotomic operator, as its definition is not as constrained as for refinement and *lggs* operators. Our approach has been implemented and applied to both artificial and real-world applications. We have demonstrated that quite complex patterns can be found in a reasonable amount of time. These contributions can be related to 2 out of 5 important issues for the future of ILP, as elaborated in a recent survey [40]: “novel search methods” and “special-purpose reasoners”.

Acknowledgements We would like to acknowledge support for this project from the Biotechnology and Biological Sciences Research Council (BBSRC grant 21BEP17028).

References

[1] Abe, K., Kawasoe, S., Asai, T., Arimura, H., Arikawa, S.: Optimized Substructure Discovery for Semi-structured Data, *Principles and Practice of Knowledge Discovery in Databases*, LNCS 2431, 2002.

[2] Albert-Lorincz, H., Boulicaut, J.-F.: Mining Frequent Sequential Patterns under Regular Expressions: A Highly Adaptive Strategy for Pushing Contraints, *SIAM International Conference on Data Mining* (C. K. Daniel Barbará, Ed.), SIAM, 2003.

[3] Alt, M., Martin, F.: Generation of Efficient Interprocedural Analyzers with PAG, *Static Analysis Symp.*, LNCS 983, 1995.

[4] Apostolico, A.: String Editing and Longest Common Subsequences, in: *Handbook of Formal Languages* (G. Rozenberg, A. Salomaa, Eds.), vol. 2 Linear Modeling: Background and Application, chapter 8, Springer-Verlag, Berlin, 1997, 361–398.

[5] Badea, L.: A refinement Operator for Theories, *Inductive Logic Programming*, LNCS 2157, 2001.

[6] Badea, L., Stanciu, M.: Refinement Operators Can Be (Weakly) Perfect, *Int. Conf. Inductive Logic Programming* (S. Džeroski, P. A. Flach, Eds.), LNCS 1634, Springer, 1999.

[7] Baker, B. S., Giancarlo, R.: Longest Common Subsequence from Fragments via Sparse Dynamic Programming, *Algorithms - ESA'98, 6th Annual European Symposium*, LNCS 1461, Springer, 1998.

[8] Brachman, R. J.: On the Epistemological Status of Semantic Nets, in: *Associative Networks: Representation of Knowledge and Use of Knowledge by Examples* (N. V. Findler, Ed.), Academic Press, New York, 1979.

[9] Brāzma, A., Jonassen, I., Eidhammer, I., Gilbert, D.: *Approaches to the automatic discovery of patterns in biosequences*, Technical Report 113, Department of Informatics, University of Bergen, Norway, 1995.

[10] Brejová, B., DiMarco, C., Vinař, T., Hidalgo, S. R., Holguin, G., Patten, C.: *Finding Patterns in Biological Sequences*, Technical Report CS-2000-22, University of Waterloo, 2000.

[11] Clare, A., King, R. D.: Predicting gene function in Saccharomyces cerevisiae, *Bioinformatics*, **19**(Suppl. 2), 2003, ii42–ii49.

[12] Cohen, W. W., Hirsh, H.: Learning the CLASSIC Description Logic: Theoretical and Experimental Results, *Principles of Knowledge Representation and Reasoning: Proc. of the 4th Int. Conf.*, Morgan Kaufmann, 1994.

[13] Dehaspe, L., Raedt, L. D.: Mining association rules in multiple relations, *Int. Workshop Inductive Logic Programming* (N. Lavrač, S. Džeroski, Eds.), LNCS 1297, Springer, 1997.

[14] Dehaspe, L., Toivonen, H., King, R. D.: Finding frequent substructures in chemical compounds, *Int. Conf. Knowledge Discovery and Data Mining* (R. Agrawal, P. Stolorz, G. Piatetsky-Shapiro, Eds.), AAAI Press., 1998.

[15] Domingos, P.: The Role of Occam's Razor in Knowledge Discovery, *Data Mining and Knowledge Discovery*, **3**(4), 1999, 409–425.

[16] Fensel, D., Wiese, M.: Refinement of rule sets with JoJo, *Eu. Conf. Machine Learning* (P. Brazdil, Ed.), LNCS 667, Springer, 1993.

[17] Ferré, S., Ridoux, O.: A Logical Generalization of Formal Concept Analysis, *Int. Conf. Conceptual Structures* (G. Mineau, B. Ganter, Eds.), LNCS 1867, Springer, 2000.

[18] Ferré, S., Ridoux, O.: A Framework for Developing Embeddable Customized Logics, *Int. Work. Logic-based Program Synthesis and Transformation* (A. Pettorossi, Ed.), LNCS 2372, Springer, 2002.

[19] Ferré, S., Ridoux, O.: An Introduction to Logical Information Systems, *Information Processing & Management*, **40**(3), 2004, 383–419.

[20] Finn, V. K.: On Machine-Oriented Formalization of Plausible Reasoning in the Style of F. Backon–J.S. Mill, *Semiotika Informatika*, **20**, 1983, 35–101, In Russian.

[21] Frühwirth, T., Hanschke, P.: *Principles and Practice of Constraint Programming*, chapter 19 – Terminological Reasoning with Constraint Handling Rules, MIT Press, 1995.

[22] Fürnkranz, J.: *Separate-and-Conquer Rule Learning*, Technical Report OEFAI-TR-96-25, Austrian Research Institute for Artificial Intelligence, Schottengasse 3, A-1010 Wien, Austria, 1996.

[23] Ganter, B., Kuznetsov, S.: Formalizing Hypotheses with Concepts, *Int. Conf. Conceptual Structures* (G. Mineau, B. Ganter, Eds.), LNCS 1867, Springer, 2000.

[24] Ganter, B., Wille, R.: *Formal Concept Analysis — Mathematical Foundations*, Springer, 1999.

[25] Geamsakul, W., Matsuda, T., Yoshida, T., Motoda, H., Washio, T.: Classifier Construction by Graph-Based Induction for Graph-Structured Data, *Pacific-Asia Conf. Advances in Knowledge Discovery and Data-Mining*, LNCS 2637, Springer, 2003.

[26] Goffeau et al, A.: The Yeast Genome Directory, *Nature*, **387**, 1997, 1–105.

[27] Inokuchi, A., Washio, T., Motoda, H.: Complete Mining of Frequent Patterns from Graphs: Mining Graph Data, *Machine Learning*, **50**(3), 2003, 321–354.

[28] Jensen, D., Cohen, P.: Multiple Comparisons in Induction Algorithms, *Machine Learning*, **38**(3), 2000, 309–338.

[29] Jonassen, I.: *Efficient discovery of conserved patterns using a pattern graph*, Technical Report 118, Department of Informatics, University of Bergen, Norway, 1996.

[30] Kononenko, I., Kovačič, M.: Learning as optimization: Stochastic generation of multiple knowledge, *Int. Workshop Machine Learning* (D. Sleeman, P. Edwards, Eds.), Morgan Kaufmann, 1992.

[31] Lanckriet, G. R., Deng, M., Cristianini, N., Jordan, M. I., Noble, W. S.: Kernel-based data fusion and its application to protein function prediction in Yeast, *Pacific Symp. Biocomputing*, 2004.

[32] Lee, S. D., Raedt, L. D.: Constraint Based Mining of First-Order Sequences in SeqLog, *Proc. of the Workshop on Multi-Relational Data Mining* (S. Džeroski, L. D. Raedt, S. Wrobel, Eds.), University of Alberta, Edmonton, Canada, July 2002.

[33] Lesh, N., Zaki, M. J., Ogihara, M.: Scalable Feature Mining for Sequential Data, *IEEE Intelligent Systems*, **15**(2), 2000, 48–56.

[34] Michie, D., Muggleton, S., Page, D., Srinivasan, A.: To the International Computing Community: a new East-West Challenge, Oxford University Computing Laboratory, Oxford, UK, 1994.

[35] Mitchell, T.: *Machine Learning*, McGraw-Hill, 1997.

[36] Muggleton, S.: Inverse Entailment and Progol, *New Generation Computation*, **13**, 1995, 245–286.

[37] Muggleton, S., Feng, C.: Efficient Induction in Logic Programs, in: *Inductive Logic Programming* (S. Muggleton, Ed.), Academic Press, 1992, 281–298.

[38] Muggleton, S., Raedt, L. D.: Inductive Logic Programming: Theory and Methods, *Journal of Logic Programming*, **19,20**, 1994, 629–679.

[39] Ouali, M., King, R. D.: Cascaded multiple classifiers for secondary structure prediction, *Prot. Sci*, **9**, 2000, 1162–1176.

[40] Page, D., Srinivasan, A.: ILP: A Short Look Back and a Longer Look Forward, *Journal of Machine Learning Research*, **4**, 2003, 415–430.

[41] Plotkin, G.: *Automatic Methods of Inductive Inference*, Ph.D. Thesis, Edinburgh University, august 1971.

[42] Quinlan, J. R.: *C4.5: programs for Machine Learning*, Machine Learning, Morgan Kaufmann, 1993.

[43] Rigoutsos, I., Floratos, A.: Combinatorial pattern discovery in biological sequences: the TEIRESIAS algorithm, *Bioinformatics*, **14**(1), 1998, 55–67.

[44] Rückert, U., Kramer, S.: Generalized Version Space Trees, *Int. Workshop Knowledge Discovery in Inductive Databases* (J.-F. Boulicaut, S. Džeroski, Eds.), On-line proceedings at `http://www.cinq-project.org/ecmlpkdd2003/`, 2003.

[45] Srinivasan, A.: Aleph: A Learning Engine for Proposing Hypotheses, Url: `http://web.comlab.ox.ac.uk/oucl/research/areas/machlearn/Aleph/`.

[46] Srinivasan, A., King, R. D.: Feature construction with Inductive Logic Programming: A study of quantitative predictions of biological activity aided by structural attributes, *Data Mining and Knowledge Discovery*, **3**(1), 1999, 37–57.

[47] Srinivasan, A., Muggleton, S., King, R.: Comparing the use of background knowledge by Inductive Logic Programming systems, *Int. Workshop on Inductive Logic Programming* (L. De Raedt, Ed.), Department of Computer Science, Katholieke Universiteit Leuven, 1995.

[48] Suppl. in Proteins: Structure, Function and Genetics: *Critical Assessment of Techniques for Protein Structure Prediction (CASP)*, vol. 45, 2001.

[49] Widmer, G.: Combining knowledge-based and instance-based learning to exploit qualitative knowledge, *Informatica*, **17**, 1993, 371–385, Special issue on Multistrategy Learning.

[50] Wille, R.: *Ordered Sets*, chapter Restructuring lattice theory: an approach based on hierarchies of concepts, Reidel, 1982, 445–470.

[51] Wolpert, D. H., Macready, W. G.: *No Free Lunch Theorems for Search*, Technical Report SFI-TR-95-02-010, Santa Fe, NM, 1995.

[52] Zucker, J.-D., Ganascia, J.-G.: Representation Changes for Efficient Learning in Structural Domains, *Int. Conf. Machine Learning* (L. Saitta, Ed.), Morgan Kaufmann, 1996.