Czech Technical University in Prague

Faculty of Electrical Engineering

Doctoral Thesis

July 2013 Ing. et ing. Petr Buryan

Czech Technical University in Prague

Faculty of Electrical Engineering
Department of Cybernetics

Refinement Action-Based Framework For Utilization Of Softcomputing In Inductive Learning

Doctoral Thesis

Ing. et ing. Petr Buryan

Prague, July 2013

Ph.D. Programme: Electrical Engineering and Information Technology
Branch of study: *Artificial Intelligence and Biocybernetics*

Supervisor: ***doc. ing. Filip Železný, PhD.***
Supervisor-Specialist: ***ing. Jiří Kubalík, PhD.***

Acknowledgments

I gratefully acknowledge the help and support generously provided by my supervisor Filip Železný and supervisor-specialist Jiří Kubalík.

Abstract

In this thesis we describe a novel approach to application of Evolutionary Algorithms (EAs) into the field of Inductive Logic Programming (ILP). One of the main issues of ILP is the time complexity that comes out of the large size of ILP search space. Improving the search efficiency is therefore the main objective of this thesis. To reach this, EAs were used in this work as they have proven to be efficient solvers for many optimization problems in many applications before.

The target of the thesis is to design a system that would use EAs to speed up the search process in ILP while also enabling to use full potential of ILP including the possibilities of first order logic as well as the refinement operators. In the same time we aim at developing a system that would not be too problem-specific and would be both user and implementation friendly.

Unlike the traditional approaches that focus on evolving populations of logical clauses, our refinement-based approach uses the evolutionary optimization process to search for sequences of refinement actions that iteratively adapt the initial working clause so that it changes into a well classifying clause. Utilization of context-sensitive refinements (adaptations) helps the search operations to produce mostly syntactically correct concepts and enables using available background knowledge both for efficiently restricting the search space and for directing the search.

To efficiently use the context information about the dataset we propose a specific structure to represent it and also an algorithm that automatically induces this structure from data prior to the start of the learning phase. The fact that the user is not required to manually define the language bias by writing down the semantic and syntactic restrictions speeds up the whole process of solving ILP tasks and gives our system the user friendliness that is needed for applying this one system to various ILP problems.

In general, the EA-based ILP system presented in this work is more flexible, less problem-specific and the framework can be easily used with any stochastic search algorithm within the ILP domain. Experimental results on several data sets verify the usefulness of this approach. The benchmarking shows that our system is capable to achieve similar or even better results than other state-of-the-art ILP systems.

Abstrakt

V této práci se věnujeme novému přístupu využití technik Evolučních Algoritmů (EA) pro řešení problémů v rámci Induktivního Logického Programování (ILP). Jedním z hlavních problémů ILP je časová náročnost řešení úloh vyplývající ze značné rozsáhlosti prohledávaného prostoru. Zefektivnění prohledávání je tak hlavní motivací této práce. Jako prostředek byly zvoleny EA, protože se již v mnoha předešlých případech ukázaly být efektivním nástrojem pro prohledávání ve velkých stavových prostorech. Jejich použití v ILP nicméně není zcela jednoduché, zejména vzhledem k rozdílnosti ILP a úloh, pro které byly EA primárně určeny.

Cílem práce je navrhnout systém, který by použil EA k urychlení řešení ILP úloh a zároveň nebyl příliš úzce zaměřený a umožnil využít všech možností ILP poskytovaných jazykem predikátové logiky prvního řádu (FOL) i refinement operátory. Současně je naší snahou vyvinout systém, který nebude příliš úzce zaměřen na úzkou skupinu dat a bude také uživatelsky i implementačně přívětivý.

Na rozdíl od ostatních aplikací EA v ILP pracuje náš přístup vždy s jednou klauzulí popsanou přímo v FOL a využívá evoluční optimalizace k jejímu upravování. Pomocí EA náš algoritmus iterativně hledá takové sekvence refinement operátorů, kterými upraví pracovní klauzuli tak, aby se zlepšila její kvalita posuzovaná zejména z hlediska klasifikační přesnosti. Kromě tohoto náš přístup využívá i kontextové informace v podobě syntaktických a sémantických restrikcí lokálně omezujících použití jednotlivých refinement operátorů. Práce přímo v FOL vede k jednoduché reprezentaci problému a dovoluje jednoduše využít dostupnou doménovou znalost (background knowledge). Díky kontextu je pak systém schopen soustředit prohledávání na syntakticky správné koncepty.

Pro efektivní využití kontextové znalosti při prohledávání jsme navíc v naší práci navrhli specifickou strukturu jejího uchování a algoritmus, který je schopen sestavit tuto strukturu z dostupných dat ještě před vlastním začátkem ILP hledání. Skutečnost, že syntaktická i sémantická omezení jsou automaticky indukována z dat bez nutnosti jejich manuální definice uživatelem, urychluje celý proces řešení ILP problému a zároveň poskytuje našemu systému dostatečnou flexibilitu pro použití v různých ILP úlohách.

Výsledky experimentů provedených s několika ILP problémy potvrdily užitečnost tohoto přístupu a srovnání s výsledky jiných prací ukázalo, že náš algoritmus je schopen dosáhnout stejných či dokonce lepších výsledků než jiné state□of□the□art algoritmy. Navržený přístup tak lze využít k efektivnímu řešení ILP úloh. Kromě toho, že je plně schopen využít možností FOL, je i univerzálnější, snadno může být použit k řešení různých ILP problémů a zároveň umožňuje jednoduše využít i jiné stochastické algoritmy než EA.

Contents

1 Introduction

In this thesis we focus on learning relational concepts expressed in First Order Logic (FOL) by means of Evolutionary algorithms. The field that focuses on this type of tasks is known as Inductive Logic Programming (ILP). In ILP the standard task is to induce a classification model that would explain given set of examples defined in FOL usually combined with the background knowledge – a set of supplementary problem information in FOL that is often available along with the examples data. The advantage of using FOL for the problem representation is its expressivity and the ability to easily represent almost any problem. However, this expressive language offers actually so much of freedom that the classification concepts can take many forms. Thereby, FOL and becomes also the source of complexity of ILP tasks as the expressivity eventually renders the task to find the right concept difficult to solve.
As a result, ILP needs to solve two nested NP-hard problems (Gotlob, 1999) – the search in huge space of concepts in FOL and the quality testing of each candidate solution. The first issue - the complexity of search for the best concept - motivated us to this work in which we try to implement nature-inspired Evolutionary algorithms (EAs) to solve the ILP tasks. However, the second issue of quality testing cannot be neglected either. Assessing candidate concepts is also very time demanding as it requires determining which examples the candidate concept covers requires search for proper matching of literals, variables and constants of the clause to each example in the data.

1.1 Motivations and Goals

The main motivation for using EAs in ILP is to face the complexity of the ILP tasks. Although there already are several ILP systems that try to use EAs to solve the difficult ILP tasks the approaches taken by these systems shows several deficiencies. The motivation to realize a new approach that would use EAs for ILP tasks was to tackle following major problems connected with application of EAs into ILP:

1) **Simple problem representation.** All current EA-based ILP searchers use specialized representation of the clauses to be able to solve the task by the EA system (see e.g. Divina, 2006; Tamaddoni-Nezhad, 2002). These representations do not often fully allow using all capabilities of ILP refinements and additionally require development of specialized search operators. This leads to the fact that the EAs for ILP are hybrid systems highly specialized on specific ILP problems and datasets which stands in contrast to one of the basic characteristics of EAs – the domain independency.
2) **Full utilization of all ILP capabilities.** Most of the EA-based ILP searchers use restricted version of ILP to solve the ILP problems which leads to limited flexibility (user defined clause templates, restriction on the type of refinement operators available) – see e..g (Neri and Saitta, 1995; Reiser P. and Riddle, 2001). An approach is needed that would enable utilization of all ILP refinement operators (and also going beyond FOL with e.g. treatment of numeric values).
3) **Efficient search for reasonable concepts.** The standard 'blind' evolutionary operators construct mainly candidate concepts that are either too general (cover all given training examples) or are meaningless in the problem domain (semantically invalid, do not cover any example); this issue is additionally amplified by specific characteristics of ILP domains where the presence of meaningful concepts among the rest of theoretically possible ones is sparse (Reiser, 1999). Solution used by standard ILP algorithms (Quinlan, 1990; Muggleton, 1995) to search the lattice of clauses ordered by subsumption and keep focus on cases covering at least some examples cannot be used for EAs. Therefore, a method is needed that would be suitable for EAs and would efficiently focus the search towards semantically and syntactically correct concepts.
4) **Refinement of the discovered concepts.** Despite the fact that EAs perform well at locating good solutions in complex search spaces, they are poor at refining these solutions (Reiser, 1999). The approach should enable to efficiently refine and optimize potential preliminary solutions.
5) **Easy utilization of additional domain knowledge.** Generally, EAs perform well in the absence of problem domain knowledge, but when such knowledge is available, it is difficult for EAs to exploit it either in the representation or in the evolutionary search (Tamaddoni-Nezhad, 2002). EAs and ILP are on opposite sides in the classification of learning processes - while EAs are known as empirical or DK-poor, ILP could be considered as DK-intensive method in this classification. EA-based learner for the ILP domain should be able to use any domain knowledge available for the problem at hand.

1.2 Contribution

To reach the goals of the Thesis we have developed a novel approach to utilization of EAs in the field of ILP. It is based on the framework of the ***Prototype Optimization with Evolved Improvement Steps*** algorithm POEMS (Kubalik and Faigl, 2006) that represents a novel and suitable alternative for discovering ILP concepts. In contrast to current approaches that focus on evolving the concepts we used the stochastic optimization process of EAs to iteratively adapt initial concepts (in the form of logical clauses) by means of context-sensitive clause refinement actions. Our algorithm works directly with clauses defined in FOL and alters them by means of sequences of ILP refinement operators that are searched for by evolutionary algorithm. Thereby, it modifies the initial concept so that it evolves towards the required conditions represented by given concept evaluation function (e.g. the entropy function).

Such implementation does not need development of any special problem-specific data structures and/or new complicated search operators like we can see among other similar systems. It also does not necessarily need any user-defined strong search and language bias (e.g. there is no necessary limit on clause form, length or on number of literals used) and offers an easy option to reach beyond FOL (e.g. by using numeric constraints). Direct access to the ILP refinement operators and direct operations with clauses represented in FOL allows using full capabilities of ILP including incorporation of the domain knowledge. Moreover, utilization of the context-sensitivity in the refinement actions helps the algorithm to focus on syntactically correct concepts and to further refine them without creating abundant number of concepts with no support in the data.

We can see **five main contributions** of this thesis:

1) *Adaptation of EA-based POEMS algorithm for the ILP domain.* Creation of POEMS-based searcher into ILP domain links EAs directly with all standard ILP refinement operators (such as replacing value with constant or removing predicate from clause). In common approaches that combine EA with ILP, this would disrupt the object-level GA in ILP. In our case this is solved by simple implementation of the set of refinement actions directly in the FOL representation.
2) *Development of framework of context-sensitive refinement actions.* Utilization of context information during stochastic clause refinement efficiently enables the algorithm to focus on the syntactically and semantically correct clauses. This in effect improves search abilities of the resulting algorithm as it limits the search space and enables to focus on meaningful clauses. The information about the context restrictions is induced automatically prior to start of the learning process and requires no special user input (see automated context induction described in contribution 3 below).
3) *Development of algorithm for automated context induction.* We proposed an algorithm that runs prior to the start of the learning phase and automatically induces the context information from the data and saves it in a structure that can be easily used by the given set of refinement actions. The fact that the user is not required to manually define the language bias by writing down the semantic and syntactic restrictions (like in other grammar-based systems) speeds up the whole process of solving ILP tasks and gives our system the user friendliness that is needed for using it to new problems.
4) *Extension of refinement actions with library based refinements produced by graph mining algorithm.* The set of refinement actions used by the EA searcher can be easily extended by clause fragments supplied either by user or by automated pattern generator. This improves the search results as well as shows flexibility of our POEMS-based system.
5) *Plateau facing technique based on complexity-preferring adjustment of the evaluation function.* We suggested facing the platteau problem by enriching the searcher's evaluation function with a complexity factor that – in contrast to standard methods – prefers more complex patterns to the smaller ones. Accompanied by clause generalization step that is

performed prior to adding the clause to the model this brings positive results as the search algorithm has higher probability to transfer the platteau and reach optimal solution behind it.

The results of experiments on both synthetic and real-world data show that the proposed approach represents a competitive alternative to the current techniques. In our experiments, where we used well-known non-trivial induction tasks, we have shown that this approach is able to construct relevant concepts both in artificial and real-world data and the obtained results were better or comparable to the results of other evolutionary stochastic systems reported in literature. We also showed that this approach is able to handle and incorporate the domain knowledge and use it during the search process that is guided by standard genetic algorithm.

1.3 Outline of the Thesis

This thesis is structured as follows. The Chapters 2 to 4 contain an introductory text, detailing the concepts and techniques relevant for utilization of evolutionary algorithms in ILP. Chapter 2 defines major concepts of Relational learning. Chapter 3 focuses on ILP, gives basic definitions and introduces standard ILP solving approaches. Chapter 4 presents the techniques of Evolutionary Algorithms and especially focuses on the latest approaches to implementing evolutionary learners in ILP. Chapter 5 introduces the POEMS algorithm as an alternative to standard approaches that does not need to translate the problem into some specific EA-suitable representation. Chapter 6 describes in detail the implementation of POEMS to ILP tasks, Chapter 7 gives experimental evaluation of the behavior of the POEMS-based system. Chapter 8 describes an extension of our system by utilization of automatically generated library refinements. Chapter 9 summarizes, gives ideas for potential further research and concludes the thesis.

1.4 Related Work

The results of the thesis as well as partial solutions and ideas used in it were published in several academic journals and conferences. In this sub-section we provide an overview of the publications related to our work. The *approach developed in this thesis* was published as a journal article in:

- Petr Buryan, Jiří Kubalík. (2011). *Context-Sensitive Refinements for Stochastic Optimisation Algorithms in Inductive Logic Programming*. Artificial Intelligence Review, vol. 35, no. 1, pp. 19-36. *(IF 1.213, http://www.springer.com/computer/ai/journal/10462)*

The *seed ideas* for this thesis and the subsequent elaboration of topics related to using refinement actions within the field of Evolutionary Algorithms and application of this principle to Relational Learning or resulted in these four conference articles:

- Petr Buryan. (2008). *Accelerating Frequent Subgraph Search by Detecting Low Support Structures.* Inductive Logic Programming, 18th International Conference.
- Petr Buryan, Jiří Kubalík, Katsumi Inoue. (2009). *Grammatical Concept Representation for Randomised Optimisation Algorithms in Relational Learning.* Proceeding of the ISDA'2009 conference. pp.1450-1455.

- Petr Buryan, Jiří Kubalík. (2010). *Context-sensitive refinements for stochastic optimization algorithms in inductive logic programming.* Proceedings of the GECCO'2010 conference, pp.1071-1072.
- Jiří Kubalík, Petr Buryan, Libor Wagner. (2010). *Solving the DNA fragment assembly problem efficiently using iterative optimization with evolved hypermutations.* Proceeding of the GECCO'2010 conference. pp.213-214.

Several *partial solutions* that were used in this thesis originated prior to the interest in using Evolutionary Algorithms for the tasks of Relational Learning. The author of the thesis studied the effect of implementing the principles of Evolutionary algorithms in GMDH networks - the principles used also in this thesis include e.g. the effects of evolutionary selection principles on quality of resulting model or the plateau facing techniques. Apart from several conference articles, following journal articles resulted from this work:

- Petr Buryan, Godfrey C. Onwubolu. (2011). *Design of Enhanced MIA-GMDH Learning Networks.* Int. J. Systems Science, vol. 42, no. 4, pp. 673-693. *(IF = 0.948, http://www.tandf.co.uk/journals/tf/00207721.html)*
- Godfrey C. Onwubolu, Petr Buryan and Frank Lemke. (2008). *Modeling Tool Wear In End-Milling Using Enhanced GMDH Learning Networks.* The International Journal of Advanced Manufacturing Technology, vol. 39, no. 11, pp. 1080-1092. *(IF = 1.103 - http://www.springer.com/engineering/production+eng/journal/170)*
- Taušer Josef and Buryan Petr (2011). *Exchange Rate Predictions in International Financial Management by Enhanced GMDH Algorithm*, Prague Economic Papers 3/2011 *(IF 0.390 - http://www.vse.cz/bulletin/555)*
- Buryan Petr and Taušer Josef. (2010). *Strojové Učení a Modelování Měnových Kurzů v Praxi Finančního Řízení.* Ekonomický časopis, 56 :781–799, 2008. *(IF 0.289 - 2010 JCR Social Science Edition)*

2 Relational Learning

In this work we study the use of Machine Learning (ML) techniques in Relational domains i.e. computer performed learning techniques to create models of relational data. Since its origination, the field of ML focused on analyzing mainly single-table attribute value datasets and its use on relational datasets that contain several data tables still remains a challenging task.
In this chapter we provide basic introduction of Relational Learning as a subfield of Machine Learning and compare it to the most frequently used ML subfield – Attribute Value Learning. The chapter is structured as follows. In the first sections we introduce the field of Machine Learning, briefly describe the fields of Attribute Value Learning and Relational Learning (RL) and compare them. Finally we conclude with several examples of problems and applications from the Relational Learning domain to give basic overview and motivation to solve RL tasks.

2.1 Machine Learning

The field of Machine Learning into which Relational Learning belongs deals with discovering information from large datasets and presenting it in an easily comprehensible format (i.e.: graphical or statistical). The strongest value added of ML is its contribution in automatically gaining knowledge from datasets that would be difficult to analyze without automated methods due to their size and complexity.
Most Machine Learning systems take as input a dataset from the analyzed problem domain that is organized as a set of examples – *observations.* From these observations they induce knowledge that is not explicitly available in the dataset. The process of knowledge discovery usually focuses on searching for patterns in the data that show statistically interesting characteristics. Following two definitions can be used to describe the task of Machine Learning:

Definition 2.1 (Learning). Learning is constructing or modifying representations of what is being experienced. (Michalski, 1980)

Definition 2.2 (Machine Learning). A computer program is said to learn from experience *E* with respect to some class of tasks *T* and performance measure *P*, if its performance at tasks in *T* as measured by *P* improves with *E* (Mitchell, 1985).

The target of the ML systems is to generalize over the input examples to construct a model of the domain (either explicit or implicit). This model can then be used e.g. to make predictions on new, previously unseen, examples. Generally, ML domain can be divided into two distinct areas – supervised learning and unsupervised learning. The unsupervised learning methods like clustering do not have the opportunity to use any information on assignment of examples to classes e.g. for estimating the quality of the model (by some score function to obtain error or reward assessment). On the contrary, supervised learning algorithms that are also subject of this thesis use such information to develop models that fit best the given data.

Definition 2.3 (Supervised Machine Learning). Given the learning task *T*, a set of classified observations (examples) E_C of the unknown target concept that are drawn from language of examples L_E and a *score(E,M)* function which specifies the quality of the model *M* with respect to the data *E* and expresses this as a real number, the goal is to find such model *M* (presented in language L_M) for which the value of score function over some unseen examples E_{test} would be optimal, i.e.

$$M^* = \arg\max_{M \subset L_M} score(E_{test}, M) \quad .$$

The search for the model usually takes the form of traversal through the search space that represents a set of all feasible solutions (or partial solutions) and has defined neighborhood structure (i.e. there is a neighborhood relation defined between some solutions that enables the learner to systematically traverse the space). When constructing the model, the learning algorithm in ML (or an *induction* algorithm) forms it as concept description and induces it from the available example dataset. The analyzed dataset is actually the knowledge available and may be represented differently from one algorithm to another. For example, the C4.5 algorithm (Quinlan, 1986) represents knowledge as decision trees, Inductive Logic Programming algorithms (Muggleton, 1991) represent knowledge in the form of a set of logical clauses etc.
According to the type of information with which the ML systems operate, kind of models they construct and the algorithms they use many types of techniques can be distinguished. The most common techniques take as input a set of examples defined as feature arrays that all have fixed same length - in an *attribute-value representation*. Database table is usually used to store these examples, each row recording one example. These datasets that can be saved in one single table are called *propositional* or *attribute valued data*. The field focusing on mining knowledge from these datasets is known as *Propositional Data Mining* or *Attribute Value Learning* and covers majority of ML techniques used in data mining applications. In contrast to this approach stands the concept of *Relational Learning* that operates with data where the observations initially cannot be expressed in the form of a single array.

2.2 Attribute-value learning

Attribute Value Learning (AVL) or *Propositional Data Mining* comprises most spread Machine Learning methods that work with datasets in which each example is expressed as a single unified element. An example is e.g. the purchasing history of a single client of a shop. These data can be stored in the form of arrays in which each array represents one purchased item and has the same elements (code of item, price, date of purchase etc.).

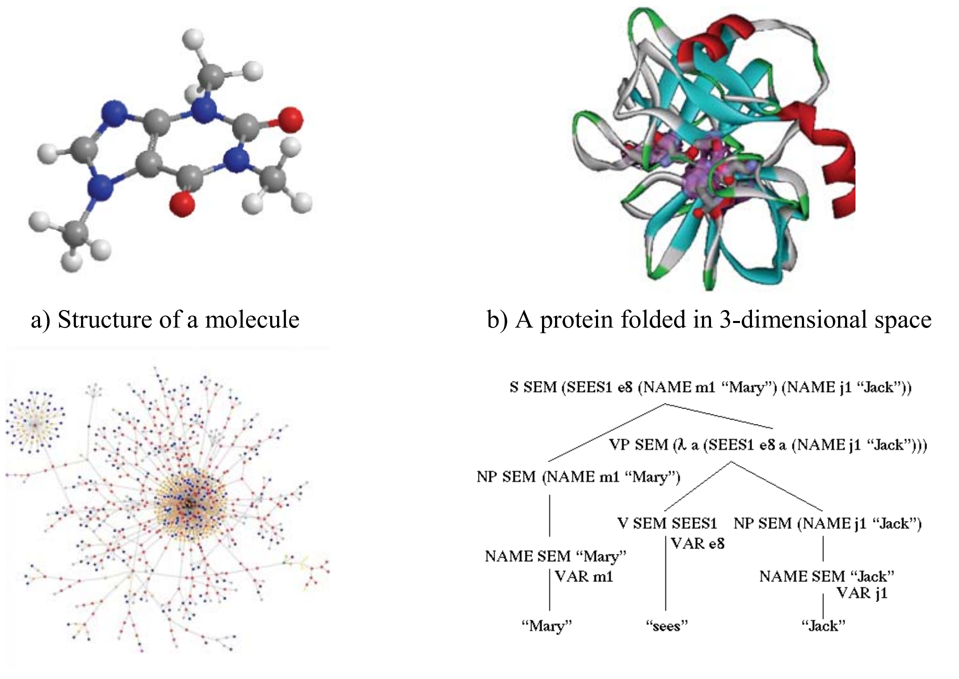
Definition 2.4 (Attribute-value Learning). Attribute-value learning is a task of Machine Learning where the examples are represented in an attribute-value representation: all taking the form of an array of observations with exactly *n* attributes **(**$n \in N_0$**).**

The algorithms in this formalism will typically employ some form of propositional logic to identify subgroups (hence the other name *Propositional Data Mining*). The attribute-value learning techniques have been used in numerous applications. They fit perfectly to the most common problems where the available information consists of observations that are statistically independent (or close to it) and each observation can be described with a fixed set of attributes.
However, in applications where more complex data structures are present and single table representation is not sufficient, AVL techniques reach their limits. The recent boom of IT has evoked development of many new and complex domains where the problem description is so complex that it has brought many problems in which the expressivity of propositional datasets is simply insufficient. A popular example of such field is sociology with its recently famous phenomenon of social networks that consist of individuals and their interconnections (which could be business relationships or kinship or trust, etc.). Other examples include computational biology or natural language understanding. The data collected in these new areas cannot be of a unified single-type array form. It is structured and extremely complex. Because the instances are variously interconnected and the datasets need to be stored and modeled through use of relational databases the datasets are called *Relational datasets* (see Figure 2.1 for several examples of these data). During time, two streams developed to handle these datasets – specific ML subfield of Relational Learning methods focusing on designing new algorithms to handle these datasets and an approach trying to use existing AVL methods called propositionalization.

2.2.1 Propositionalization

The main idea of *propositionalization* is that constructing propositional features from structured data enables to employ some fast, robust and well-known methods of AVL. First the initial RL problem is reformulated into an attribute-value by describing it in a tabular representation induced by the algorithm and then some efficient AVL method is used to mine this transformed dataset.
The work (DeRaedt, 1998) shows that it is possible to convert the simplest form of Inductive Logic Programming (one of ML subfields focusing on relational data), into a propositionalized form. Yet, the same work (DeRaedt 1998) also points out that such transformations need time demanding computational efforts. In ILP, creating the tabular form leads to creating a new reformulated dataset that can be of exponential size as well as highly redundant. Due to this, for many ILP problems the set

cannot be, in the end, easily directly addressed by AVL techniques. This technique of solving RL problem is used by a number of RL systems. Examples are LINUS system of (Lavrac and Dzeroski, 1994) that first introduced the idea of propositionalization, REPART system (Zucker & Ganascia, 1998) or RSD (Zelezny and Lavrac, 2006).



a) Structure of a molecule

b) A protein folded in 3-dimensional space

d) Webpage structure

d) Semantic tree for sentence '*Mary sees Jack*'

Figure 2.1 – Four examples of relational data

The advantage of propositionalization is that it capitalizes on vast portfolio of efficient AVL techniques with all its positive aspects. Today, the main motivation if transforming the RL task into less complex AVL form is to allow solving the RL problem faster by existing efficient AVL algorithm. From the point of view of this thesis, propositionalization is also interesting from another angle – it is often used as a method to implement Evolutionary Algorithms into the field of ILP. The reason is that EAs are efficient in searching for optimal combinations in large single-table databases.

2.3 Relational Learning

The relational datasets brought new challenges for ML techniques. Not only, that it is difficult to make standard AVL techniques work with structured datasets (through propozitionalization) but the new form of data representation brought in also other problems. The most important is the increase in computational complexity. Consider mining "*simple*" association rules from single attribute value table by AVL algorithm that take e.g. following form:

if (weather=sunny and wind=none) then play_tennis=true.

This was already shown to be NP-complete problem (Morishita, 1998). The extension to structured data (or even introduction of variables) renders the whole task almost intractable already for standard-sized problems. Therefore, current ML research aims towards developing new techniques of handling with Relational Data that are called *Relational Learning*.
Relational Learning (RL) can be perceived as an extension of Attribute Value Learning techniques to handle data with complex relations among attributes and in some cases with relations among the examples. RL deals with problem domains where the data cannot be recorded in the form of single table and structures like graphs are natural and the only possible problem representation (see examples depicted in Figure 2.1 below).

Definition 2.5 (Relational Learning). Relational Learning is a task of Machine Learning where the examples are represented in the form of complex relational structures: the dataset is built from set of attributes A and set of n-ary relations R defined between attributes from A. Each example is defined as $e = \{\{a\}^p, \{r\}^q\} \mid a \in \boldsymbol{A}, r \in \boldsymbol{R}, p,q \in \boldsymbol{N_0}\}$, i.e. examples consist of a various number of attributes and various number of relations between these attributes (De Raedt, 2008).

In effect, RL represents more complex issue than AVL as the search space is much larger. Actually, AVL problems represent a special subset of RL problems in which there is only one relation with fixed number of n attributes defined. The main differences of RL to standard AVL include:

- ***Data complexity.*** RL deals with large amounts of data of high dimensionality – the search space grows “very fast” with the amount of information provided by the examples and also by additional problem information (called background knowledge – see bullet point below);
- ***Data uniformity.*** The individual examples do not carry standard uniform set of attributes of fixed size and therefore cannot be expressed as entities in multidimensional space; although on the general level there exists semantic concept of the data (the dataset can be stored in a form of multi-table database) each example may consist of parts that may be appear more than once, be incomplete or be absent in some examples (relational approach enables that the provided information may easily come from different sources);
- ***Model comprehensibility.*** Relational representation brings the issue of presenting the learned knowledge in a way that enables insight on the rules of the phenomena that produced the data;
- ***Utilization of numerical values.*** Generally, RL can combine relational data with numerical computations;
- ***Background knowledge utilization.*** RL often accepts extra information about the problem domain which is often used to incorporate expert knowledge as a part of the so called background knowledge - the expert can choose e.g. the granularity of the model elements (the level of abstraction) by providing pieces of different granularity (e.g. in chemical problems it is possible to chose between constructing models from atoms, chemical structures, properties of whole molecule, etc.).

2.3.1 ILP and Graph Mining: Two Important RL Frameworks

Based on the expressivity of the language, two main approaches exist for mining information from relational datasets - logic-based approaches and graph-based approaches (see e.g. (Ketkar et al. 2005) for comparison):

- **Inductive Logic Programming (ILP).** Being the first truly relational approach, ILP originally came from the intersection of Machine Learning with Logic Programming (Muggleton, 1991) and as such aims on finding patterns expressed as logic programs in first order logic (FOL). Initially ILP focused on automated program induction from examples that consist of collection of facts defined in FOL solving the task formulated as a binary classification. However, these days ILP is used also for other problems than "basic" classification – e.g. clustering, association analysis etc.
- **Graph Mining (GM).** Graphs are less expressive than FOL clauses, yet provide a sufficient and natural representation for many problems with structured data. The typical database for Graph Mining (Cook and Holder, 2007) task consists of labeled graphs, and search algorithm is used to search for interesting substructures (subgraphs) that may or may not be present. Although the Graph Mining differs from the logic-based approaches in several ways, the most obvious difference is the syntax of the representation.

Comparing the two approaches, ILP allows for expression of more complicated patterns that can include variables and constraints among variables or even functions.

No matter how expressive the language is, both domains deal with one common complexity issue – difficulty of pattern evaluation (Dantsin et al, 2001). All RL algorithms work on the '*generate and test*' principle (at least to some extent) and they need to search a large space of combinations that may form the pattern. Quality of patterns is judged by frequency of its occurrence in the data (pattern coverage). Therefore the covering test is the most critical operation as any pattern that is discovered during the search process must be tested as to its coverage (how many examples from the dataset include the pattern). This is a typical combinatorial problem and its solving is time demanding. For coverage calculations GM uses *subgraph isomorphism* tests, ILP uses *θ-subsumption* (see next chapter) which are both NP-complete problems (Gottlob 1999). Apart from the structure of the dataset, this complexity is another reason why specific algorithms need to be developed for RL tasks.

There are two general techniques to solving RL tasks that have been historically used both in GM and in ILP:

- *Direct approaches* that focus on development of special methods for the original representation and solve directly RL problem.
- *Indirect approaches* seeking techniques to bridge the gap between RL and classical AVL by transforming relational data into simpler propositional form and subsequently using the AVL learning algorithms.

The "*direct*" approach takes advantage of fitting the new method to the expressivity of the data representation that enables to construct complex features for its models for the price of heavy time and

memory requirements. These methods will be better described in the next chapters of this work (see the subsection below for the description of GM, see Chapter 3 for ILP description). The "*indirect*" approach on the other hand searches for methods to combine current AVL algorithms with RL tasks by using propositionalization and was described earlier.
In the following subsections we will briefly introduce the area of Graph Mining and conclude with presenting some examples of RL applications. ILP as the main topic of this thesis is described later in the subsequent Chapter 3.

2.3.2 Graphs and Graph Mining

In this subsection we will briefly introduce the field of Graph Mining a relatively recent and quickly developing subfield of RL focusing on mining from *graph data* i.e. data that are structured as graphs. Currently, the GM domain focuses mainly on the following tasks:

– finding *interesting patterns* in the graphs,
– finding *groups of similar graphs* (clustering),
– building *predictive models* for the graphs (classification).

Except these three points, the area of graph mining offers vast portfolio of different tasks e.g. graph similarity (Cook and Holder, 2006) measurement, anomaly detections (Noble and Cook, 2003) and many others. However, the most common task in graph mining is the frequent subgraph search where the point is to find frequently occurring subgraphs in a given collection of graphs and use these for various purposes. Apart from frequent subgraph search, several applications exist focusing on mining specific pattern types from graph databases (Vanetik et al. 2002) etc.

Definition 2.6 (Graph). A graph or undirected graph G (Cook and Holder, 2007) is an ordered pair of sets $G = (V, E)$, where:

- V is a set of vertices or nodes;
- E is a (unordered) set of pairs of distinct vertices from V, called edges.

A labeled graph is a graph with labels assigned to its nodes and edges i.e. a 5-tuple

$$G = (V, E, \lambda, \Sigma_E, \Sigma_V),$$

where

$$E \subseteq V \times \Sigma_E \times V,$$
$$\lambda : V \rightarrow \Sigma_V,$$

Σ_V being the set of vertex labels (vertex label alphabet),
Σ_E is the alphabet of edge labels and
λ *is* the labeling function that maps vertices and edges to their labels.
All sets are generally assumed to be finite, unless explicitly stated otherwise

Different nodes or edges can carry the same label. Molecule structure represents an example of a labeled graph, where each atom represents a vertex in graph labeled by name of the element and each bond stands for an edge labeled by the bond's valence.
A subgraph G_s of graph G is itself also a graph defined as

$$G_s = (V_s, E_s),$$

where $V_s \subseteq V$ *and* $E_s \subseteq E$, e.g. it consists only of edges of graph G in V that connect only the nodes in V while both edges and vertices of G_S have same labels as in G.
Affirming a graph G is a subgraph of H means finding mutually matching pairs of both vertices and edges in these structures. Let $V(G)$ be the vertex set of a graph and $E(G)$ its edge set. Graphs G and H are subisomorphic *iff* there is an injection $f\colon V(G) \rightarrow V(H)$, such that $(u,a,v) \in E(G)$ if and only if $(f(u), a, f(v)) \in E(H)$.
To reformulate in other words, a graph G is sub-isomorphic to graph H *iff* graph G is isomorphic to at least one subgraph of H.
Generally, there are three main sources of complexity in the RL domains. These three bottlenecks that are in slightly adjusted form shared by all RL domains are pattern frequency testing, pattern equality testing and search and generation of potentially useful patterns. For graph mining, these take the following forms (Valiant, 1983):

1) The subgraph isomorphism problem (NP-complete) - determining if a graph is a part of another (operation used when determining coverage of given subgraph pattern);
2) The graph isomorphism problem (GI complexity – in general neither P nor NP) - determining if two graphs are equivalent (this operation is used e.g. when comparing two discovered fragments as isomorphic sub-graphs are actually the same sub-graph);
3) The search in the lattice of frequent subgraph fragments (generally NP-complete) (Mitchell, 1997).

It is not only that solutions of problems 1) and 2) are complex and time demanding but these tasks have to be solved often during the GM search. In addition, the search spaces that are determined by the most of real graph databases are sparse meaning that only a small fraction from all possible subgraphs really occurs within the databases. As a consequence, large amount of patterns generated during the search turns out to have zero coverage and search by means of random sampling becomes very difficult.

2.3.2.1 Frequent Subgraph Mining Algorithms

In the following text we will briefly describe the field of frequent subgraph mining and briefly describe the latest state-of-the-art GM algorithm Gaston (this algorithm was used for specific purposes also in this thesis). Despite the flexibility of data representation offered by the graph structures specialized data mining approaches targeted this domain as late as in mid 1990s. This is caused mainly by to the complexity of search for frequent subgraphs as mentioned above. The novel approaches were developed on the basis extended of classical basic frequent fragment mining approach - complete subgraph lattice traversal utilized in the 1970s (Ullman, 1976). Current state-of-the-art miners apply one of two basic search strategies – the *Apriori-like* strategy or the *pattern extension* strategy.
The *Apriori based techniques*, inspired by Apriori algorithm (Inokuchi et al., 2001) (market basket analysis), search for frequent itemsets using a "bottom up" approach and breadth-first search - starting from set of most frequent fragments where each fragment is a single vertex, at each step they take the set of all already discovered fragments (each having k nodes) and generate candidate fragment with k +1 nodes as their extensions and combinations. The algorithms AGM (Inokuchi et al., 2001), FSG

(Kuramochi and Karypis, 2001), and FFSM (Huan et al. 2003) may be seen as the major representatives.
Algorithms that use the *pattern extension strategy* also search the subgraph lattice in steps. In each step they extend the previously discovered patterns by new node and/or edge. Following algorithms stand for the state-of-the-art in this domain: Subdue (Cook and Holder, 2006), MoFa (Borgelt and Berthold, 2002), gSpan (Yan and Han, 2002) and Gaston (Nijssen and Kok, 2005).

2.3.3 Relational Learning – Examples of Applications

In the following text we will briefly describe several applications of RL to give motivation to solve the RL problems. Relational (especially graph) models are popularly used in many applications where complex data need to be processed (e.g. marketing analysis, social networks, web analysis and many others). However, the main domains in which full expressivity of graph relational structures are employed are currently cheminformatics and biology. Not only that the information these work with is relational from its nature but they currently dispose of highly available and accessible experimental data (free accessible databases *Kyoto Encyclopaedia of Genes and Genomes* (KEGG) and many others) that support further fast development of graph mining in the domains.

2.3.3.1 Applications in Chemistry

Currently very popular *cheminformatics* tasks use relational learning in the drug discovery and compound synthesis. Motivated by both commercial and scientific reasons, attention is paid to searches for frequent molecular fragments within chemical compounds with specific biological activity (Helma et al. 2004). These patterns are thereafter used for discovery and explanation of beneficial effects at different doses and limiting deleterious effects as well as for more efficient drug production, cheaper tests or even discovery of new active compounds not covered by existing patents.
Cheminformatics actually represents the first domain in which both ILP and GM become popular mainly due to research connected with the Warmr (King, Srinivasan and Dehaspe, 2001) algorithm. A proof that the area seems to be promising is to be seen in the fact that specialized algorithms for this area have been developed (such as e.g. the *MoFa* algorithm (Borgelt, Berthold, 2002) focused on molecular databases and biochemical questions). The research in the previous decade has shown that RL can construct accurate rules which predict the chemical activity of tested compounds (e.g. Srinivasan 2001, Lendwehre et al. 2010).
An example of a large area focusing on using Relational Learning (both ILP and Graph Mining) is Structure-Activity Relationship (SAR) focusing on discovering models of drug activity (Richard, 1999). These studies generally assume that the set of given examples (compounds) can be divided into two subsets: active and inactive. The task is to search the examples for patterns that can be used to differentiate the active versus the inactive compounds. The following information is most often available for these compounds:

– structural information - atoms and bonds in the compound, information on specific atom-bond groups (nitro- group, ketone- group etc.);
– chemo-physical properties such as molecular weight etc.

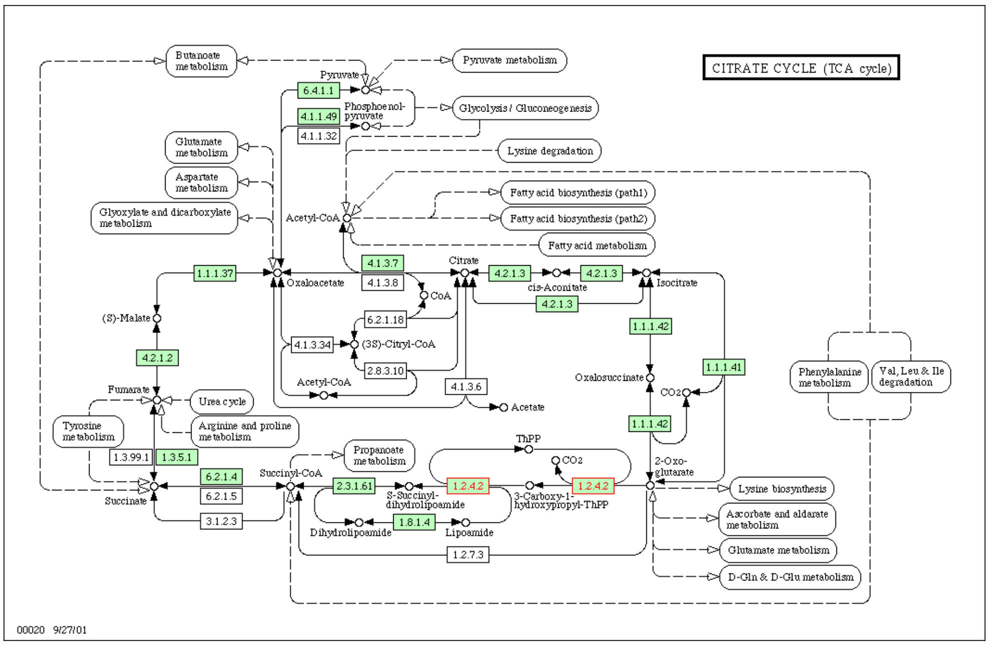


Figure 2.2 – Example of a metabolic pathway – citrate cycle (source: KEGG database)

2.3.3.2 Applications in Biology

An example of area that deals with extremely complex data in biology is *biochemistry and molecular biology*. This field deals with biological networks, graphs that represent highly relational structures describing various biomolecular interactions. Generally one can distinguish three basic categories of biological relational datasets - metabolic networks e.g. enzymatic processes creating energy and other parts of the cell - see e.g. (Lacroix et al. 2008), protein networks (protein-protein interactions implementing signal communications (Martin et al. 2007)) and genetic networks (regulation of DNA - protein gene expression analysis (Stetter et al. 2003)).
Analyses of these biomolecular interaction data focus again mainly on discovering common sequences and motifs with the target to understand mainly these phenomena:

- motifs of cellular interactions;
- evolutionary relationships;
- differences between networks in different organisms;
- patterns of gene regulation.

Several mining tasks are followed in the bioinformatics coming out from given targets:

- unsupervised learning (e.g. Ligtenberg et al. 2009)
    - patterns in several networks in one biological species;
    - patterns in one network across several species.
- supervised learning (e.g. Zhang and Cui, 2010)
    - distinguish networks in one species from those in another species;
    - distinguish one network from another network across several species.

An example of an interesting problem that is highly challenging in gene relevance networks analyses (Huang et al. 2004) is the prevalence of spurious interactions due to self-activators, abundant protein contaminants and weak, non-specific integrations.

2.3.3.3 Applications in Business

Standard applications of RL in business focus like many other ML algorithms on discovering regularities in financial time series (e.g. Kovalerchuk et al. 1998). Yet there are also other problems that are solved. An interesting example of application, that focuses directly on utilization of RL for purposes of revenue improvement comes from marketing and is directly related to social network mapping and analyses (Small world phenomena (Watts, 1999)). The idea is called viral marketing (Domingos and Richardson, 2001) that is based on the idea that considering social word-of-mouth effects might lower marketing costs. As many current marketing approaches it abandons the traditional unfocused (mass) marketing and turns towards direct marketing. In this form of marketing, first an attempt is made to select the most profitable customers and then market only to them. Instead of considering each person in isolation, viral marketing reflects the effects of one person's buying decisions on his/her neighbors in the social network.
Another interesting application lies in the cross-market customer segmentations (Pei et al. 2005). In a specific market, the similarity among customers in market behavior can be modeled as a similarity graph. Each customer is a vertex in the graph and two customers are connected by an edge with label corresponding to the similarity of their behaviors in the market. This improved segmentation in effect enables optimized customer service and communication in effect leading to revenue optimization.

2.3.3.4 Other applications

An example of application from different domain is program code optimization. In size optimizing compilers (e.g. x86 Assembler), an isolation of parts of the program that occur multiple times might lead to smaller code. Currently used technique of Procedural Abstraction is based on the sequential program code. However, a code of a program (or its parts) may be represented also by a directed graph (data flow graph, DFG) that in addition expresses the dependencies among the instructions (Muchnick 1997). Frequent fragments of this graph could be later isolated and used for optimizing the size of the code and its performance.
Subject of research that is close to this thesis and should be mentioned are graph databases (Cook and Hoder, 2007). These represent source of information for further processing and efficient accessibility

of the data is therefore very important. While a relational database maintains different instances of the same relationship structure (represented by its ER schema) a graph database maintains different relationship structures. It should allow (along with standard attribute queries) also processing of the structural queries – queries over the relationship structure itself. (e.g. structural similarity, substructure, template matching, etc.). In graph databases, structure matching has to be performed against a set of graphs and therefore proper storage, pre-processing and indexing of structures are crucial and requires development of special techniques.

2.3.4 Reference Literature on RL

Describing all concepts and applications is simply out of scope of this work. Other domains of interest include web searching (analyses of XML databases, web structure, etc.), and many other (CAD schemes of electrical circuits, program control flow, traffic flow, workflow analysis, etc.). In case when the reader would like to reach out for more information much more detailed description of all RL-related topics is available (among other texts) we state here several reference works. For ILP, good and comprehensive introduction to ILP can be found e.g. in (Muggleton 1991; Lavrac and Dzeroski, 1994) or in De Raedt's book (DeRaedt, 2008). A lot of information can be gained from the collection edited by Dzeroski and Lavrac (Dzeroski and Lavrac 2001) on relational data mining. Regarding the Graph Mining domain, Cook and Holder (Cook and Hoder, 2007) is a good starting point to learning from graphs. A good overview about the algorithms in GM may be found e.g. in (Washio and Motoda, 2003; Nijssen and Kok, 2006). For other domains of RL, Dzeroski and Lavrac (Dzeroski and Lavrac 2001) is also a good source for reading about multi-relational data mining and multi-relational data mining systems.

2.4 Summary

In this chapter we have introduced the field of Machine Learning as well as its two sub-domains – Attribute Value Learning (AVL) and Relational Learning (RL). AVL techniques represent historically older area of Machine Learning research that focuses on dealing with data that can be described a single table. RL is on the other hand a younger domain of Machine Learning. Its development is driven by increasing amount of more complex datasets that come from expanding research in new areas like computational biology where one needs to processes data that cannot be compressed into structurally unified single-table dataset. In these cases the data are of relational nature as they usually describe relationships between this instance and other data elements or its parts.

RL includes two main subdomains that differ in the expressivity of the description language used – Inductive Logic Programming (ILP) and Graph Mining (GM). While GM deals with data structured as graphs (data described by language that uses only unary and binary relations), ILP uses higher expressivity of FOL in inducing models from sets of logical clauses (allowing data to be flexibly described by relations with more than two attributes as in graphs). After describing briefly these two main domains of RL – Inductive Logic Programming and Graph Mining - we concluded the chapter with few motivational application areas of RL. The next chapter is devoted to the main subject of this thesis, Inductive Logic Programming.

3 Inductive Logic Programming

In this chapter we give a brief introduction to ILP. The purpose of this chapter is to give some basic definitions of FOL that are needed later on in this thesis. This section is structured as follows. After introducing the field of ILP as well as the main sources of its complexity in the first section we continue the chapter with defining basic terms. After describing standard setting of an ILP task we address the problem of solving the task including ordering and traversal of space of concepts and verifying if a given clause covers an example. In the last two sections we give briefly describe three standard well known ILP systems and finalize the chapter with basic extensions of ILP search by simple stochastic search approaches.

3.1 Introduction

Inductive Logic Programming as a subfield of Relational Learning was originally defined as the intersection of Machine Learning and Logic Programming (Muggleton 1991). Although the scope of the research field expanded and now ILP enables to combine numerical and statistical models with relational data, the basic problem definition remains the same. To solve a problem in the Inductive Logic Programming means to construct a theory that would well explain given classified examples defined it in first order logic (FOL). The process of theory construction uses given examples and supplemental knowledge to reach the explanation in an inductive manner - by reversing the process of deductive reasoning.

Inductive reasoning is actually quite common form of everyday reasoning. While deductive reasoning derives specific consequences *E* from a prior theory *T*, inductive reasoning actually derives a general theory *T* from specific beliefs *E.* For example, if *T* says that all kinds of snakes have no legs, *E* might state that some given particular snake species (e.g. a python) has no legs. In inductive reasoning the process starts with considering several particular examples *E* of snake species (pythons, cobras and other) combined with observation that these have no legs. Based on these examples, *T* might take a form of an implication stating that if an object is a snake than it has no legs (equivalent to *"all snakes have no legs"*). An advantage of ILP is that the induced rules and models are typically comprehensible enough to be understood by humans and can be easily inspected to determine the acquired principles.

In standard ILP problems the situation is more complicated and the search for the theory takes the form of a search in a space of candidate theories (Muggleton 1991) expressed in a given representation language (FOL). Using FOL, the search space is specifically defined by two sources of input data - classified examples and background theory (supplemental information coming from the problem definition and other potentially useful information for the model construction). Starting from some initial theory, generalization and specialization operators may be applied to scan the space of all theories and direct the search towards "better evaluated" theories. In a standard setting, better theories are those that explain many positive and few negative examples.

The advantages of ILP are mainly comprehensibility of the constructed models and straightforward possibility to use information on complex structures (graphs, for example) in Machine Learning (ML). Although ILP should aspire to be one of frequent ML methods because many real-life machine-learning problems lend themselves naturally to a FOL representation, its wider utilization keeps being inhibited by the complexity of ILP as solving problem in ILP actually means solving an extremely complex problem. (Gottlob 1997) showed that the simple bounded ILP consistency problem is $\sum_2$-complete. This is one class higher in the polynomial hierarchy than NP-complete (or $\sum_1$-complete) problems. This complexity comes from two main sources:

- **The complexity of search for a clause is NP-hard** – clauses are logical disjunctions with their elements being predicates, constants and variables. The search for the optimal clause is a combinatorial problem that is solved in the search space defined by the possible combinations. Search in such exponentially large space is NP-hard (not mentioning that the number of equivalent clauses is theoretically indefinite) (Dantsin et al. 2001).
- **The complexity of theory quality evaluation is NP-hard** – ILP learners work in the large search space on the *generate-and-test* principle and as such they need frequent evaluation of clauses. Answering the question which training examples are covered by a clause is the prerequisite to assess the quality of the clause (compute the heuristic value), an essential part in the search for a good clause. Yet, simply proving an ordinary clause takes much time as theorem proving is an NP-hard problem in general (Gottlob 1999, Cook 1971).

Although in this thesis we focus on efficiently approaching the issue of search complexity, we still need to partially address the issue of efficient evaluations as these two issues go hand in hand in ILP.

ILP is using first-order logic as the problem describing language. The motivation is it allows us to flexibly define objects and their properties and relations. The syntax of first-order logic has two different categories: *terms* and *formulas*. Terms are constructed from *constants*, *variables* and *function symbols*. Atomic formulas are constructed by filling in terms in the argument places of predicate symbols. Usually, while terms are used to define "objects" formulas are used to represent object properties or relations. The necessary definitions of the language as well as and other necessary terms used in ILP are given in the next section.

3.2 Basic ILP Definitions and Principles

In this subsection we provide the basic definitions needed for proper introduction and description of ILP. The complete set of definitions on Inductive Logic Programming may be found e.g. in (Lavrac and Dzeroski, 1994) or (Muggleton, 1991). Unless explicitly stated otherwise, in the definitions we consider finite alphabets of function symbols, constant, variable and predicate symbols.

Definition 3.1 (Terms, atoms, literals and constants). A term is either a constant, a variable or a function symbol (functor) followed by n terms between parentheses, where $n \in \boldsymbol{N}$ is the arity of the functor. A constant is a functor of arity zero.

Let P be an n-ary function symbol and $t_1, \dots t_n$ logical terms. Then $P(t_1, \dots t_n)$ is an atom, $t_1, \dots t_n$ are called the arguments of the atom. Literal is an atom or a negation of an atom (positive or negative literals). For example, $P(a, b)$ is a positive literal, which is true if $P(a, b)$ is true, while $\neg P(a, b)$ is a negative literal, which is true if $P(a, b)$ is false.

Definition 3.2 (Formula). Let A and B be formulas and X a variable. A formula is defined either as a literal, negation of a formula $\neg A$, a conjunction of formulas $A \wedge B$, a disjunction of formulas $A \vee B$, an implication $A \Rightarrow B$ or $A \rightarrow B$, an equivalence $A \Leftrightarrow B$, an existential quantifier $\exists X\, A$ or a universal quantifier $\forall X\, A$. Common praxis is to use the universal quantifier implicitly: if no quantifier is given for a variable, we assume the variable is universally quantified. A formula is ground formula when it contains no variables.

Definition 3.3 (Clause). A clause is a finite disjunction of zero or more literals where all variables are universally quantified. A disjunction of literals is a set of literals. Thus the clause $C = \{a_1, a_2,.. \neg a_i, \neg a_{i+1}, .. \neg a_n\}$, where $a_1, .., a_n$ are literals, can be equivalently represented as $C = (a_1 \vee a_2 \vee .. \neg a_i \vee \neg a_{i+1} \vee .. \neg a_n)$. All the variables in a clause are implicitly universally quantified. In the field of logic programming it is common to write such clause as an implication where we first write the consequence followed by the antecedent: $(b_1 \wedge b_2 \wedge .. b_n) \Rightarrow a$. The consequence a is called the '*head*' of the clause, the antecedent is called the '*body*' of the clause.

Definition 3.4 (Fact). A fact is a clause with no literals in the body.

Definition 3.5 (Horn Clause). A Horn clause h is a set of literals (implicitly representing a disjunction) with at most one positive literal. It takes the following form:

$$h \Leftarrow l_1 \wedge l_2 \wedge \dots l_n,$$

where h and the l_i are logical atoms, and $l_1,\dots, l_n$ is called body of the clause.

Definition 3.6 (Theory). A theory is a conjunction of Horn clauses. In this thesis we often use the word '*hypothesis'* for a (finite) theory that was created by an algorithm. Another word used for theory is Logic Program.

Definition 3.7 (Coverage). The theory T covers an example e *iff* $B \wedge T \models e$ where '$\models$' is a symbol meaning logical entailment (consequence). In other words, the theory T explains (i.e. logically entails) the example e with respect to the background knowledge B. Detailed definition of entailment can be found e.g. in (Lavrac and Dzeroski, 1994)

Definition 3.8 (Closed world assumption). The closed world assumption, introduced by Reiter (Reiter, 1978), says that if a literal L cannot be inferred from a given set of background knowledge B then L is not satisfiable under B, i.e. $B \not\models L$ (theory B cannot explain L).

Definition 3.9 (Substitution). A substitution is a set of bindings of the form: *V/c* where c is a constant and V is a variable. For example, $\{A/a,B/b\}$ is a substitution which binds the variable A to the constant a and the variable B to the constant b. We use the notation $T\theta$ to denote the result of binding all the variables in the term T according to the substitution θ. For example, suppose $T = f(A,B)$ and $\theta = \{A/a,B/b\}$, then $T\theta = f(a, b)$.

Definition 3.10 (θ-subsumption). A clause C θ-subsumes ($\preccurlyeq_\theta$) a clause D if there exists a substitution θ such that $C\theta \subseteq D$. For example, the clause C: $f(A,B) \Leftarrow p(B,G), q(G,A)$ θ-subsumes the clause D: $f(a, b) \Leftarrow p(b, g), q(g, a), t(a, d)$ by the substitution $\theta=\{A/a,B/b,G/g\}$. In such case $C \preccurlyeq_\theta D$.
The reason why θ-subsumption is mentioned in this section is that it is used for coverage calculations to evaluate quality of solutions in ILP instead of logical implication (Robinson, 1965). While implication is undecidable in general, the θ-subsumption, that represents an incomplete approximation to logical implication, is a NP-complete problem (Kapur and Narendran 1986). Generally speaking, solving the θ-subsumption problem means solving following task: being given two clauses, C and D, the task is to find a substitution θ such that all literals of C, via this substitution θ, are contained in the set of the literals of D.

3.3 Basic ILP Setting

In ILP, the problem domain is described in FOL which provides the formal framework for concept description and further reasoning. The task of ILP is to induce concept description from the input of background knowledge and set of classified examples. The user-supplied examples are classified into finite number of classes and each example is defined as a set of ground facts. The concepts take form of hypotheses. For simplification, in the further text we will assume only single-clause hypotheses (multiple-clause hypotheses can be also built, when necessary, e.g. by a greedy set covering algorithm used in an iterative manner). The basic setting of ILP (Muggleton 1991) is defined in a following way.

Definition 3.11 (Basic Setting of ILP). Given a problem defined as triplet (B, E^+, E^-), where B is set of information called background knowledge (in the form of a set of definite clauses), E^+ and E^- are sets of positive (E^+) and negative (E^-) examples that are not directly derivable from B, the basic task is to find a theory T in the form of a logic program, such that T is complete and consistent, i.e.:

1. *for all* $e_p \in E^+$: $B \wedge T \models e_p$ (T is complete);

2. *for all* $e_n \in E^-$: $B \wedge T \not\models e_n$ (T is consistent);

where both B and T are sets of function-free first-order Horn clauses.

Informally speaking, the aim of ILP is to find a theory T such that T covers every positive example e_p from E^+ and T does not cover any negative example e_n from E^-. Such theory is called *correct* (i.e. complete with respect to E^+ and consistent with respect to E^-). In ILP, the theory is learned by

searching in a space of hypotheses, where a hypothesis is a possible logic program in FOL. The close world assumption is used in ILP to limit the size and number definitions needed in background knowledge to sufficiently describe the problem.
Usually, Prolog notation is used to express the examples, background knowledge and also the theory.

3.3.1 Prolog Notation

In ILP, the examples are given to the learner in terms of objects and object relations instead of just simple propositions, the example description language being FOL. Many ILP systems are implemented in *Prolog* language (PROgramming with LOGic) and, subsequently, many examples in the ILP literature are given its notation. Therefore, also in this work a flattened *Prolog* (PROgramming with LOGic) language notation is followed.
The basic elements of logic programming are functions, constants and variables – in Prolog notation, names of variables are strings of symbols that start with a capital letter or an underscore. To distinguish constants from variables, names of constants, function symbols and predicates are strings starting with a lower case letter. Arguments of predicates can be either variables (starting with an upper-case character) or constants (starting with a lower-case character).

Example 3.1 – The Prolog notation. *p(X1, c), q(X2)* is a conjunction that consists of two literals – the first literal *p(X1, b)* formed by the binary predicate *p* of arity 2, sometimes referred to as *p/2*, with 2 arguments: the variable *X1* and the constant *c,* and the second literal *q(X2)* consisting of the unary predicate *q* of arity 1 (referred to as *q/1*) with one argument - the variable *X2*.

3.3.2 ILP - An Example

Let us consider an example in Figure 3.1. This example is inspired by standard Bongard (Bongard 1970) problem set. Each example of the problem consists of several geometrical objects, the task is to determine one or more patterns (i.e. particular set of objects, their shapes and their placement) that can be used to correctly discriminate the positive examples from the negative ones.

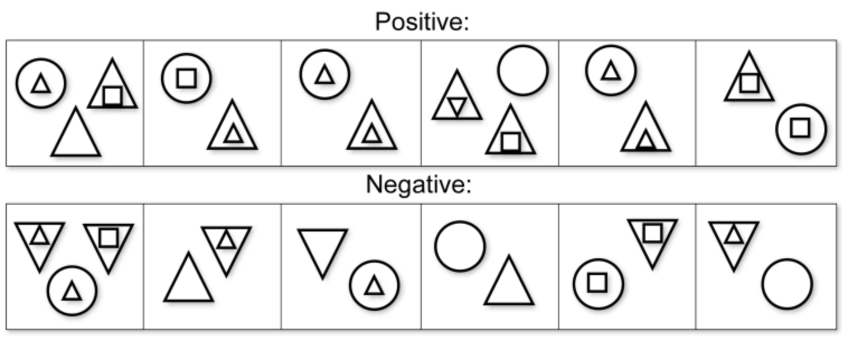


Figure 3.1 - An ILP example - Bongard problem

This task actually means to build a concept of the example pictures in the upper row so that no picture from the lower row is covered by this concept. The Bongard problems are difficult to solve with attribute-value learning but can be approached quite naturally using a relational representation. Learning of these concepts actually represents a nontrivial problem even for a human solver, not considering a computer.
By means of FOL, the first positive example in the upper row of Figure 3.1 can be represented as follows:

$$group\_of\_shapes(e1) \wedge circle(c1) \wedge square(s1) \wedge triangle(t1,up) \wedge triangle(t2,up) \wedge triangle(t3,up) \wedge inside(t3,c1) \wedge inside(s1,t2) \wedge in\_group(e1,c1) \wedge in\_group(e1,t1) \wedge in\_group(e1,t2) \wedge in\_group(e1,t3) \wedge in\_group(e1,s1) \rightarrow positive(e1).$$

Considering all examples, following concept covers all positive instances (instances in the upper row):

$$group\_of\_shapes(X) \wedge in\_group(X,Y1) \wedge triangle(Y1,up) \wedge inside(Y1,Z) \rightarrow positive(X).$$

This means that a group of objects (an example) *X* is classified as positive if it contains two objects *Y1* and *Z* where *Y1* is an upward oriented triangle that contains some other object *Z* inside. Note that in ILP, a symbol starting with capital letter is usually treated as variable while all symbols that start with lower case letters are treated as constants. Another example of a concept may be the following clause:

$$group(X) \wedge in\_group(X,Y1) \wedge triangle(Y1,down) \rightarrow negative(X).$$

This classifies as negative all objects that contain some downward oriented triangle. However, this concept is not consistent - it covers all negative examples (second row) but takes in also one positive example.
Although being non-trivial, the Bongard-like problems represent only a small niche of issues solved by ILP. Many other types of problems emerged in the ILP domain that are not concerned only with the standard ILP task to learn set of classification rules. Among others, we can see ILP solving e.g. issues of induction of logical decision trees (Blockeel and De Raedt, 1998), first-order regression (Karalic and Bratko, 1997), learning of clausal theories and association rules (De Raedt 1997), first-order clustering (Blockeel and De Raedt, 1997), frequent subgroup discovery (Wrobel 1997) or many other (see e.g. Flach and Lavrac, 2002).

3.4 Solving standard ILP Tasks

The process of discriminative theory induction can be seen as a search in the space of concepts (or candidate theories) with the aim to find a correct concept with respect to the examples given. In fact, the search for interesting concept in ILP is in principle just another example of optimization problem. Solving optimization tasks means for a given problem instance, find a solution (combination, ordering, or assignment) with maximal (or minimal) value of the defined objective function. From this point of view, we search for a theory (model) that would optimally classify the set of given examples i.e. would cover as much positive and as few negative examples as possible. As such, the ILP problem from Definition 3.11 can be also defined in a following way (Landwehr et al. 2007):

Definition 3.12 (**ILP task as optimization problem).** Given background theory *B*, a set of classified examples *E*, a language of clauses *L* that specifies the clauses that are allowed in the theory and a *score(E,T,B)* function, which specifies the quality of the theory *T* with respect to the data *E* and prefers complete and consistent theories and the background theory *B*, the task is to find a theory T^* such that

$$T^* = \arg\max_{T \subset L} score(E,T,B)\ .$$

The quality of *T* (its score) may be judged by several criteria that usually reflect the accuracy of *T* on given training set and most often are a combination of completeness, consistency, and simplicity of theory *T* (Muggleton 1991).

3.4.1 Basic Search in ILP

As mentioned, ILP solves a search problem in which a space of candidate solutions is searched to find the best solution *T** with respect to given evaluation function characterizing the quality of the solutions to the ILP problem. The search is performed by means of generalization or specialization refinement operators that are applied on some initial theory.
The simplest approach to the problem is to solve it with a naïve generate and test algorithm. However, such algorithm is computationally too expensive to be of practical interest. Therefore, different techniques based on different approaches have been developed to efficiently structure and scan the solution space to allow for pruning the space and guiding the search so that good solutions can be found efficiently.

Algorithm 3.1 - Generic ILP algorithm

```
Input: set of classified examples E, set of clause refinement operators
Output: theory for example classification

function ILPsearch
   Operators, background knowledge BgK, evaluation function EvalFn
   Theories = Initialize(BgK, Examples)
   repeat
      Clause = EvaluateAndSelectClause(Theories, Examples, EvalFn)
      NewClause = Refine(Clause, Examples, BgK, Operators)
      Theories = Reduce(Theories + NewClause, Examples)
   until StoppingCriterionIsMet(Theories, Examples, EvalFn)
   return(Theories)
```

Despite using different specific approaches, during the search ILP systems typically generate candidate concepts by refining either given start hypothesis or some hypothesis encountered earlier. The original hypothesis becomes parental to the new one. In standard search, the algorithm starts from an initial hypothesis and applies pre-defined operators called refinement operators to create new hypotheses. The aim is to find such hypothesis that cover many positive examples and minimum of the negative ones and thus can be used e.g. for classification. Due to simplification of the explanation only

the search for hypotheses with single clause is described in the following text (multiple-clause hypotheses can be then constructed e.g. by greedy set-covering approach). A generic version of an ILP search algorithm is given in Algorithm 3.1 (Muggleton and deRaedt, 1994).
The description of the four main functions used by the algorithm is following:

1) ***Initialize***(*BgK*, *Examples*): initializes the theory (or a set of theories) and sets them as the starting points of the search (e.g. *Theories* = {*true*} or *Theories* = *Examples*);
2) ***EvaluateAndSelectClause***(*Theories*, *Examples, EvalFn*): evaluates clauses in the theory (calculates coverages and applies the evaluation function) and selects the most promising candidate clause from the theory with respect to given set of examples *Examples* and the clause evaluation function *EvalFn;*
3) ***Refine***(*Clause*, *Examples, BgK, Operators*): applies the refinement operators (e.g. generalization or specialization) and returns refinements of the given *Clause* (creates set of "*neighbours*" of the clause)
4) ***Reduce***(*Theories + NewClause*, *Examples, EvalFn*): a function used to discard unpromising theories (dependent on search mechanism - hill-climbing, beam search, best-first search etc.)

From the description it is clear that the two critical steps are the ones that define the search strategy: evaluation the refinement (steps 2 and 3). The search space of all possible clauses is defined by the set of predicates and constants used in the dataset, the nodes of the space (clauses) are connected by links that represent possible refinements (performed by refinement operators). Within this space the search algorithm repeatedly generates refinements (modifications) of currently opened clauses by means of refinement operators and evaluates these modifications with utilization of heuristic evaluation function.
Expressive Power of ILP comes with increased computational cost as both the hypothesis search and coverage tests are NP-hard problems. Two types of bias (i.e. mechanisms used to constrain the search for hypotheses) are used in ILP to tackle this issue:

1) *Search bias* – determines factors that control how the hypothesis space is searched, how the search algorithm prefers one hypothesis to other ones (fitness function can be seen as search bias, because it biases the algorithm towards fitter hypotheses. Two parameters that define the search bias are organization and traversal through search space and the heuristic function (see next chapter).
2) *Language bias* – represents restrictions put on the hypotheses that may be created and by this determines the search space itself - imposes syntactic or semantic constraints on what kind of models can be represented by the algorithm (Van Laer 2000). Search space reduction and interesting concept exclusion are the two trade-offs that need to be considered when selecting the bias. On one hand, the bias allows tractable learning procedures; on the other hand, it limits the variety of expressible theories. The language bias determines following parameters:
   – the syntactic restrictions of the selected logic formalism,

– the vocabulary of predicate, function, variables and constant symbols: function-free clauses, ground clauses (e.g. without variables), non-recursive clauses, mode declarations (input/output behavior) of the predicates' arguments;
– numerical values treatment – the FOL is by definition discrete, therefore numerical parameters should be treated in a special way (e.g. discretization);

Language bias is often used to make the search more efficient and introduces factors like modes (define what literals can be added in the clauses), types (constraints imposed on the variables and the relations between variables), or constraints on the number of literals clauses can contain.

3.4.1.1 Search Space in ILP

Almost all search algorithms in ML domain adopt the natural idea to structure the search space by imposing an ordering on it and then traverse this lattice by means of specific search operators. In ILP, a general-to-specific ordering is usually applied to the space of hypotheses in order to structure into a lattice within which the search is performed - new hypotheses are created by means of refinement operators (Nienhuys-Cheng and de Wolf, 1997). These operators actually represent an interface between the ILP problem and the (usually general) search algorithm. Two types of refinements are recognized according to the direction these operators forward the search: specialization and generalization (see Figure 3.2).

Definition 3.13 (Refinement Operator). A *quasi-ordering* $\preccurlyeq$ is a reflexive and transitive relation. In a quasi-ordered space (S, $\preccurlyeq$) a downward refinement operator ρ is a mapping $\rho: S \rightarrow S$, such that for any $C \in S$ we have that $C' \in \rho(C)$ implies $C' \preccurlyeq C$. C' is called a specialization of C. An upward refinement operator σ is a mapping $\sigma: S \rightarrow S$, such that for any $C \in S$ we have that $C'' \in \sigma(C)$ implies $C \preccurlyeq C''$. C'' is called a generalization of C.

Specialization operator takes a clause and produces a set of specializations of the clause. Two basic types of ILP specialization operations (as used also in Figure 3.2) are

1) adding a literal into the clause body;
2) unification of two variables;
3) substitution of a variable by a constant or by a compound term.

Definition 3.14 (Specialization). A hypothesis F is a specialization of G *iff* $G \models F$ (F is a logical consequence of G, i.e. any model of G is a model of F).
Generalization is reverse operation to specialization and includes applying an inverse substitution (e.g. put variable instead of constant) and/or removing a literal from the body of the hypothesis. Considering covered examples, a hypothesis G is more general than a hypothesis F, and F is more specific than G, if all the examples covered by F are also covered by G.

Example 3.2 - Specialization refinement operators. The following definition represents the most often used specialization operator:

$$\varphi_{spec}(C) \subseteq \{ C \wedge l \mid l \in L\} \cup \{C\theta \mid \theta \text{ is a substitution}\},$$

where C is clause, l is literal, L is set of literals that are used in the language of the ILP task. Operator defined in this way offers the mentioned basic types of utilization:

– *turn variable into constant*: apply a substitution $\{X / c\}$ where c is a constant and X a variable that appears in the clause C
– *unify two variables*: apply a substitution $\{X / Y\}$ where X and Y already appear in C
– *add literal*: turn clause C into C', where $C' = C \wedge l$ and l is a literal

Considering the organization of the search space into the lattice, two basic search strategies can be distinguished. First one is the *bottom-up* approach. Systems employing this approach start the search from some very specific hypothesis and use generalization of this hypothesis during the search process. The second where approach the algorithm builds the hypothesis in a general to specific order starting with some general (e.g. empty) hypothesis and using specialization is called *top-down*.
One of advantages of searching the space of hypotheses ordered by their generality is that the specialization relation has an *anti-monotonicity* property and pruning based on this anti-monotonic property can be applied to simplify the search.

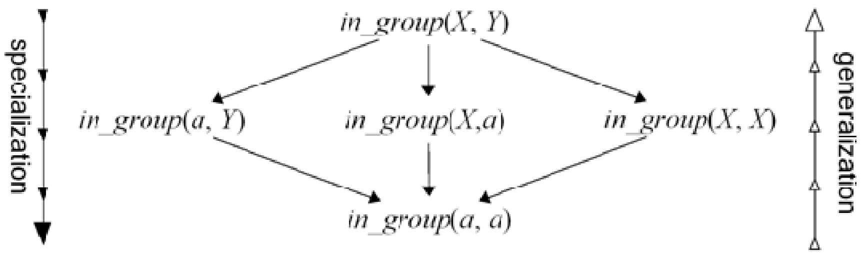


Figure 3.2 – Hypothesis lattice, generalization and specialization in ILP

Definition 3.15 (Anti-monotonic Constraint). An anti-monotonic constraint is a constraint c_n such that for all clauses C_d created by specialization from some clause C, C_d satisfy c_n if C satisfies it.
Under the anti-monotonicity c_n, when a pattern (e.g. a hypothesis H) does not satisfy $\boldsymbol{c_n}$ (the hypothesis is not frequent) then none of its specializations can satisfy $\boldsymbol{c_n}$ as well (none of clauses specialized from H are frequent either). Therefore, it becomes possible to prune huge parts of the search space which from principle cannot contain interesting patterns as we can exclude all patterns that contain the on-frequent one. This principle has been studied already by (Mitchell, 1980) within his "*learning as search"* framework. Perhaps the most famous instance of an algorithm that takes advantage of the anti-monotonic properties of the searched space is the *A priori* algorithm (Agrawal et al., 1996).
The search bias the searchers usually apply is defined by these two parameters:

– *Hypothesis lattice* traversal – determines in which direction the space is searched. Two options from which the choice can be made are top-down or bottom-up (or combination of both):
   a) *Top-down* search - systems start with an empty body (most general clause) which entails all the examples and traverses the lattice of clauses in a top down manner to find an optimal hypothesis (e.g. *FOIL* system – see below);

b) *Bottom-up* search - systems search for the target hypothesis by starting with some specific hypothesis and then traverse the lattice of clauses (either implication or subsumption) in a Bottom-up manner (e.g. *GOLEM* system);

– *Search heuristics* – the algorithm can but does not have to use the heuristic function. According to this we can see two types of search strategies:

a) *uninformed search* (depth-first, breadth-first, iterative deepening) - a rarely used approach, because of the huge hypothesis space; used e.g. by *HYPER* (Bratko 1999) algorithm during search for complete hypotheses (i.e., sets of programs clauses, not repeated search for individual clauses);

b) *heuristic search* (best-first, hill-climbing, beam search) – search that uses clause evaluation functions based on coverage calculations to direct the search towards interesting areas of the search space. Almost all ILP systems use some clause evaluation criteria both for directing and stopping the search.

3.4.1.2 Measuring Quality of Theories and Clauses

From the beginning of ILP it was clear that the complexity of the search will hinder the ILP systems from enumerating all theories for each problem. Therefore, most of the systems adopt some heuristic pruning techniques and judge the theories mainly according to their completeness, consistency, and compactness. However, there are two main aspects to clasue and theory evaluation in ILP:

1) *Coverage calculation – θ-subsumption* is used to answer the question which positive and negative training examples are covered by the given clause; NP-complexity of this issue makes it critical for any ILP algorithm to efficiently deal with coverage calculations because the search space is huge and many clauses are evaluated during the search.
2) *Evaluation function used for theory quality assessment* – the evaluation functions defines the search preferences in a form of mathematical function and might affect the search especially in cases when the definition does not match the desired target.

Theory Coverage Calculation. In ILP, used clause coverage calculations are tested by the θ-subsumption (an approximation to logical implication). Therefore, the NP complexity of θ–subsumption represents an important problem in computational logic – the fact that it lies at the core of the hypotheses coverage test and is actually the bottleneck of any ILP system makes it a particularly relevant factor to the Inductive Logic Programming (ILP) community.
The standard algorithm for solving the θ-subsumption problem is SLD-resolution. The logic of the algorithm is following: when testing the subsumption $C \preccurlyeq_{\theta} D$, all possible mappings from the literals in C onto the literals in D (for the same predicate symbol) are constructed and tested in a depth-first search manner. As such the algorithm has complexity $O(L_D{}^{L_C})$ where L_C and L_D are the lengths of clauses C and D.
Latest research on improving the coverage calculations focuses on several techniques that include query transformations (Costa 2002), query-packs (Blockeel et al. 2002), lazy evaluation of examples (Camacho 2003, Fonseca 2009) or random restarts (Železný et al. 2006).

Query transformations apply systematic techniques to reduce the number of computations necessary to determine coverage of examples focusing on redundancies in clause predicates. The transformation produces the transformed clause by processing every literal in the body of the input clause by applying intelligent cut ('*!*' in Prolog notation) and '*once*' functions to split clauses into independent partitions of subgoals (Costa 2002) and use previous results of already evaluated "*parent*" clauses. The idea behind the *query-packs* is to group similar queries into a tree-like structure. The technique of *lazy evaluation* of examples tries to avoid unnecessary use of examples in the coverage computations while keeping the completeness of the coverage calculation.

In our work we are using the latest state-of-the-art coverage calculation algorithm *RESUMER2* that use the *random restarts* technique (Kuzelka and Zelezny, 2008). This algorithm was originally designed to support efficient mining in large relational structures by enabling fast pattern evaluation and as such suits needs of EA-based ILP learner. Inspired by SAT solvers, it applies a randomized restart mechanism, where if it finds itself stuck for a long time in a subsumption test, it restarts subsumption with a different variable ordering. The basic restarted strategy is augmented by allowing certain communication between odd and even restarts without losing the exponential runtime distribution decay guarantee resulting from mutual independence of restart pairs.

Latest research has demonstrated that vast gains in efficiency can be achieved by using unorthodox subsumption algorithms as opposed to standard procedures provided e.g. by a Prolog engine. Recent statistical performance studies of search algorithms in difficult combinatorial problems have demonstrated the benefits of randomizing and restarting the search procedure. Specifically, it has been found that if the search cost distribution of the non-restarted randomized search exhibits a slower-than-exponential decay (that is, a "heavy tail"), restarts can reduce the search cost expectation. To avoid the heavy tails, we are using the latest randomized restarted subsumption testing algorithm *RESUMER2* in our work as the coverage calculator.

RESUMER2 is complete in that it correctly decides both subsumption and non-subsumption in finite time. The underlying logic of coverage calculation in *RESUMER2* is determinate matching. In the idea originally presented in (Kietz and Lobbe, 1994), where signatures (fingerprints) of a literal are computed taking into account its neighbors (i.e. variables and literals it interacts). If the same unique fingerprint exists on both clauses for a given pair of literals these can be safely matched. *RESUMER2* computes these signatures with first level neighbors. The empirically tests of *RESUMER2* against the state-of-the-art subsumption algorithm showed that it outperforms the stat-of-the-art techniques (especially for larger patterns).

The other area of coverage calculation optimization focuses on efficient query reorganizations and efficient processing of the example set. Several improvements to Prolog's clause evaluation function have been developed – the work of (Blockeel et al. 2002) considers reordering candidate clauses to reduce the number of redundant queries. In (Santos Costa et al. 2003) the authors developed several techniques for intelligently reordering terms within clauses to reduce backtracking, in (Srinivasan 199) a set of techniques for working with a large number of examples is introduced that only considers using a fraction of all available examples in the learning process, and in (Sebag and Rouveirol 2000) stochastic matching was used to perform approximate inference in polynomial (as opposed to exponential) time.

Evaluating the Theory Quality. Besides setting the language bias and structuring the search space lattice to allow for efficient search, another aspect of ILP is how to guide the search by evaluating the quality of interim candidate solutions. The evaluation criterion generally has the form of a function F: $C \rightarrow \mathbb{R}$, where C is the set of clauses defined in FOL. Some systems use gain functions in their search by means of which they compare a clause and its refinement trying to follow the steepest gradient of the function. Gain function is a mapping G: $C \times C \rightarrow \mathbb{R}$.
Most ILP searchers use similar evaluation criteria based on identifying the best clause as a clause that achieves the highest accuracy and compression, i.e. the simplest clause that can explain the largest number of positive examples (maximally close to being complete) without incorrectly explaining any negative examples (maximally close to being consistent). The three main requirements are reflected in most of the evaluation criteria used in ILP search – clause accuracy, clause novelty and clause simplicity:

$$quality(clause) = \alpha \times consistency(clause) + \beta \times completeness(clause) +$$

$$+\gamma \times simplicity(clause).$$

Following notation that is in accordance with standard data mining techniques will be used in the formulas below:

- tp_T stands for true positives (positive examples covered by theory T),
- tn_T means true negatives (negative examples not covered by T),
- fp_T stands for false positives (negative examples covered by T),
- fn_T means false negatives (positive examples not covered by T),
- len_T denotes number of literals that T contains.

Clause consistency and completeness. There are many possible functions that have been used to describe the clause quality based on its coverage. In Machine Learning and more specifically ILP, the most popular evaluation function used to evaluate the final result of learning is the predictive accuracy:

$$accuracy(T) = \frac{tp_T + tn_T}{tp_T + tn_T + fp_T + fn_T},$$

However, most of the ILP systems use different function than accuracy in their learning phase. Quality of the clauses can be measured in absolute terms or in relative terms. Examples of quality measured in absolute number of examples covered are compression or novelty. Compression is used by the well known ILP system Progol:

$$compression(T) = tp_T - fp_T - len_T + 1$$

Less frequent function is the novelty of a clause (Lavrac et al. 1999). A clause $B \rightarrow H$ is novel if H and B are statistically independent i.e. the rule cannot be inferred from marginal frequencies $n(H)$ and $n(B)$ where $n(H) = tp_T + fn_T$ is the number of positive examples and $n(B) = tp_T + fp_T$ is the number of examples covered by the body of the rule. Novelty is therefore defined as relative difference between observed probability of H and B co-occurrence defined as $p(H \wedge B) = \frac{tp_T}{N}$, where N is total number of examples, and the expected number of co-occurrences if H and B would be independent:

$$novelty(T) = \frac{tp_T}{N} - \frac{tp_T + fn_T}{N} \cdot \frac{tp_T + fp_T}{N}, \quad N = tp_T + tn_T + fp_T + fn_T$$

Another standard option is to use a criterion that comes out of the basic measures - precision and recall that measure the quality of clauses in relative terms. Recall is a function of examples correctly classified by T (true positives) and examples misclassified by T (false negatives). Precision is a function of true positives and examples misclassified as positives (false positives). The definitions of both measures are as follows:

$$precision\,(T) = \frac{tp_T}{tp_T + fp_T}, \quad recall(T) = \frac{tp_T}{tp_T + fn_T}.$$

Based on these measures we can state here for example the functions of entropy, Gini index and Laplace index to be often used in heuristic search for classification rules and patterns:

$$entropy(T) = P \cdot \log(P) + (1 - P) \cdot \log(1 - P), \quad P = \frac{tp_T}{tp_T + fp_T}$$

$$gini(T) = 2 \cdot P \cdot (1 - P), \quad P = \frac{tp_T}{tp_T + fp_T}$$

$$laplace(T) = \frac{tp_T + 1}{tp_T + fp_T + 2}$$

An interesting option that allows setting balance between recall and precision is the *F-measure* (Sokolova et al. 2006). *F-measure* of theory T is defined as

$$F(T) = \frac{(\beta^2 + 1) \cdot precision\,(T) \cdot recall\,(T)}{\beta^2 \cdot precision\,(T) + recall\,(T)}.$$

From this definition it is clear that by means of the β parameter it is possible to control the effect of both precision and recall. The *F-measure* is evenly balanced when $\beta^2 = 1$, it favors precision when $\beta^2 > 1$ and recall otherwise. This measure was already used in ILP (Mamer et al. 2008) to solve the grammar induction problem. However, the opportunity to fine tune the precision-recall balance also means that the user needs to spend time properly setting the β parameter.

Other measures can be found e.g. in (Cussens 1993 - pseudo-Bayes conditional probability, Lavrac et al. 1999 - weighted relative accuracy or Muggleton 1996 - Bayesian score). Special ILP cases exist apart from the described learning from positive and negative examples e.g. when learning from positive examples only. In such case the ILP algorithms usually estimate the probability distribution of positive and negative examples and assume the positive examples are gathered by drawing randomly from all the examples and discarding negative examples drawn.

Clause simplicity. Generally, there are two basic motivations why the factor of clause simplicity should be incorporated into the search.

- *The clause should have sufficient generalization ability.* Using the concept of simplicity generally comes out of the Occam's razor principle that advises to prefer simpler hypotheses that fit the data to the more complicated ones. By this principle, the risk of overfitting the training data is reduced. Several methods can be found in literature to define numerically the simplicity of a theory and incorporate it into the search criteria.
- *The acquired model should be comprehensible to human users.* Complex models that contain complex clauses may easily be perceived as black-boxes even though the rules are defined in a transparent manner.

Therefore, many ILP algorithms, such as Progol (see next chapter), use clause evaluation function that introduces bias in favor of low size-complexity. However, in contrast to the two previous parameters of completeness and consistency that are easy to define (see Definition 3.11) and many measures build directly on these definitions (see above), the clause simplicity aspect can be assessed in many ways. Following techniques can be seen as example approaches to dealing with the issue of excess clause complexity:

1) **Short rules.** The simplest rule to tackle overfitting is to measure the length of clauses and prefer shorter clauses to the longer ones. Although the length of a clause depends on the representation used, usually the length can be expressed as the number of literals that are used within the hypothesis (Muggleton 1995). The number of utilized constants or shared variables can also supplement this simplicity measure. An example of simple performance measure that includes also the simplicity factor is the function *f(C)* that is used by the popular *Progol* algorithm that is used to assess the quality of each clause *C*: $f(C) = tp_C - (fp_C + (len_C - 1) + h_C)$, where tp_C is the number of positive examples covered by *C* (completeness factor), fp_C the number of negative examples covered (consistency factor), len_C is the number of predicates in *C* (simplicity factor). Parameter h_C is the number of atoms (heuristically estimated) that still have to be added in order to complete the clause.
2) **Minimal Description Length (MDL).** The minimum description length (MDL) principle (Rissanen 1978) prefers the theories that minimize the length (complexity) of the theory and the length of the data encoded with respect to the theory. In MDL, usually the number of bits needed for encoding a model or the data is meant by the length parameter. More formally, if *T* is a theory inferred from data *D*, then the total description length is given as $DL(T,D) = DL(T) + DL(D|T)$ (all description lengths measured in bits).
3) **Information gain.** The function of information gain (Quinlan, 1986) measures the complexity by assessing how a change in a hypothesis does affect classification of the examples. The function is used e.g. by ILP system FOIL.

Although utilization of similar concepts in the search strategy to bias the search toward shorter rules is a measure to counter potential overfitting, preference of short clauses during ILP search may also bring negative effects. Unlike to the standard learning problems, following strictly the Occam's razor principle in ILP brings up an interesting problem for the heuristic search – *the plateau phenomena* (Alphonse et al. 2004).

The Plateau Phenomenon. This term is used to describe situations, when the evaluation function that is used to prioritize the nodes of the search space during the search is constant in its parts and the search in these areas goes more or less blind. An example visualization of a search space with plateau is in Figure 3.3. If the search algorithm finds itself on the plateau it needs several search steps to cross it and get to some search node where the node evaluation function *f* shows some gradient.

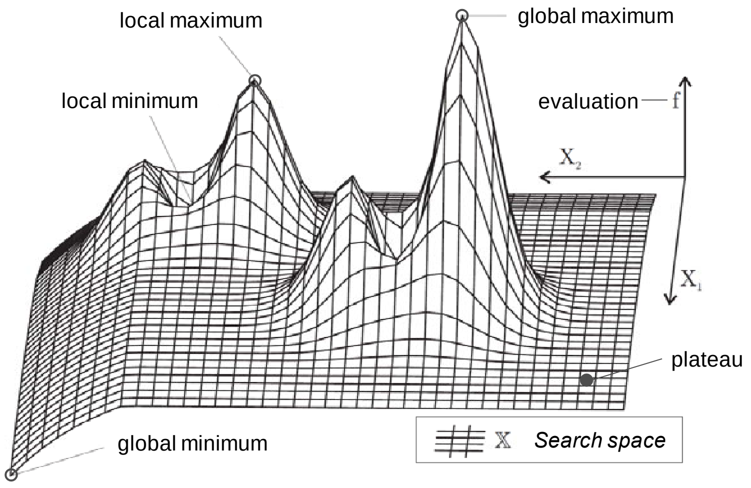


Figure 3.3 - Plateau in the search space

This definitely poses a problem for any heuristic search because the plateau has to be crossed without being able to differentiate between individual refinements. The search for a solution gets difficult because the extreme flatness of the heuristic landscape makes any search revert to a random search. In such cases simple greedy learners (like basic FOIL) cannot solve the problems optimally and in reasonable time.

An analysis and explanation of the plateau phenomenon is given in the work of (Sera et al. 2001). According to the study, the ILP covering test is NP-complete and therefore exhibits a sharp phase transition in its coverage probability. As the heuristic value of a hypothesis depends on the number of covered examples, the regions in which clauses cover either all or none of the examples represent plateaus that need to be crossed during search without any informative heuristic value. Studies on this topic consistently report (Alphonse et al.) that for all learners, there exists a failure region where the learnt theories are seemingly randomly constructed, with no better predictive accuracy than random guessing.

This problem of plateaus has been already well studied in the problem solving community (Korf 1985) and the basic solution proposed is based on macro-operators. Macro-operators represent compound refinement operators that are defined by composition of elementary refinement operators. Application of several elementary operators at a time (adding or removing several literals in ILP) showed to be more likely to cross non informative plateaus.

The hypothesis space in ILP is already difficult to search as the existence of valid concepts in it is sparse. The plateau phenomenon is another factor derogates the search process and makes learning process time demanding and more susceptible to being trapped in local optima.

3.5 Popular ILP Systems

To conclude this chapter, in the next two sections we briefly describe three well known systems for solving ILP problems and finish with basic stochastic extensions to standard ILP learning process. The ILP systems selected represent the basis of ILP learning – we briefly describe FOIL (Quinlan, 1990), Progol (Muggleton, 1995; Muggleton, 1996) and Aleph ILP learners. The selection of these systems was taken based on the fact that these represent the most popular systems for ILP and as such they were already used in a number of ILP applications.

These systems were designed for the purpose of learning of classification rules, i.e. for the is the standard ILP setting Foil is efficient and best understood, while Aleph and Progol might be considered less efficient yet have been also used in many ILP applications. While Progol uses a combined search strategy, FOIL and Aleph search the hypothesis space in a top-down manner. In addition, Progol and Aleph use inverse entailment technique that learns hypotheses by first constructing the most specific hypothesis covering a selected seed example (called bottom clause) and then search for generalization of the bottom clause with optimal value of the search heuristic. All systems mainly focus on single predicate learning from positive and negative examples and background knowledge but use different clause evaluation functions.

Algorithm 3.2 The set-covering algorithm in ILP

```
Input: Set of examples to cover $E$
      $Model$ = {};
      while(not_empty($E$)) do
            $NextClause$ = findBestPossibleClause($E$)
            $Model$ = $Model \cup \{NextClause\}$
            $E_C$ = getExamplesCoveredByClause($NextClause$)
            E = E \ $E_C$
      end
 return $Mdl$
```

The FOIL system (Quinlan, 1990) is an ILP search engine that is using a top-down approach for learning relational concepts or theories, which are represented as an ordered sequence of function-free definite clauses. Considering given background knowledge relations and (classified) examples of the target concept relation, FOIL starts with the most general theory and searches with a set-covering approach (see Algorithm 3.2) repeatedly adding a clause to the theory that embodies some of the both positive and negative cases.

Particular clauses are generated by adding literals one at a time. Each new literal added is that which increases the evaluation function information gain to maximum. Information gain is a function that compares quality of the theory extension based on examples correctly classified before and after addition of the new literal. Literals, which fall within some percentage of the maximum, can be held for later backtracking. FOIL uses also some preference toward literals that come up with new variables, but are determinate. To avoid overly-complex clauses, FOIL uses a result from the minimum description length principle despite a small number of cases, when the description length of

the clause exceeds the description length of the examples covered by the clause. FOIL has been used to a many cases of relational domains, which include not only the "standard" chemoinformatics domains (Landwehr et al. 2007) but also e.g. learning patterns in hypertext (Slattery & Craven 1998).
The Progol system (Muggleton and Feng 1990) is an ILP system that operates with inverse entailment controlled by mode declarations to learn relational concepts represented by definite clauses. Progol is actually an incremental learner which uses the set-covering approach to reach the final theory. Clauses attained from the current example are included to the set of background knowledge, and the covered positive examples are excluded from consideration. Inverse entailment identifies the most-specific clause according to the background knowledge, the current example and the mode declarations. Progol's mode declarations constrain, how variables are interconnected with literals, as in FOIL but also limits the number of instantiations of these literals within a theory. In Progol the evaluation function is enriched by heuristic estimate of the clause difference from ideally classifying clause and as such the algorithm mimics an A* search of a lattice ordered by θ-subsumption and bounded by the most-specific clause. The search is generally controlled by maximizing theory compression and by a refinement operator, which keeps the search within the lattice and averts redundancy. Considering the algorithm utilizations, Progol has achieved successes in several relational domains, e.g. protein structure problem (Turcotte et al. 2001) or mutagenesis problem (Muggleton 1995).
The ALEPH system (Srinivasan, 1999) is an ILP system realized in Prolog programming language by A. Srinivasan in the Computing Laboratory at Oxford University. The system is originally written exclusively for compilation with the YAP Prolog compiler. It implements learning algorithm similar to Progol – a sequential covering algorithm (or "separate and conquer" algorithm). The system learns clauses one by one and at individual step it removes positive examples, which are included in the clause.
The main algorithm of ALEPH comprises of four basic steps. First, the system selects some positive example to use it as the "seed" example. Then, it creates the most specific clause - the "bottom clause" - entailing the selected example. The bottom clause is constructed by conjoining all available facts about the seed example. Next step is then the search for generalizations of this bottom clause by running a general to specific search. These generalized clauses are scored according to a selected evaluation metric, and the clause with the highest score is added to the final theory. This process keeps on repeating to the point until such theory (set of clauses) is discovered, which covers every positive example. These days, ALEPH is allowing users to modify and control each of these steps to individual needs and additionally it also supports a variety of specific search algorithms including randomized search.

3.6 ILP Systems and Stochastic Search

Searching for a model in ILP represents a NP-hard problem. Theoretical and practical studies in ILP show that running general complete search without imposing strong restrictions on expected concept scheme or its refinement makes learning in ILP computationally infeasible. Randomized methods have proven extremely useful tools to search very large spaces. Therefore, current standard ILP learners employ many heuristics and speed-up techniques (Cohen 1994) including utilization of stochastic search algorithms.

For example, basic search strategies of standard ILP systems (including the most popular systems FOIL (Quinlan 1990), Aleph (Srinivasan and Camacho, 1993) and Progol (Muggleton 1995) are based on sequential set covering and beam search or hill climbing. This setting enables further search space pruning as partial ordering may be imposed on the searched space of logical clauses and the coverage *anti-monotonic* constraint can be used. Because clauses can be ordered by their generality large parts of the search space can be cut off - once the search has discovered a clause *C* with zero coverage (or coverage below some threshold) none of the clauses derived by specialization from *C* has to be considered because they cannot cover more examples than *C*. Unfortunately, this combination of set covering and greedy search makes the ILP search strongly susceptible to local optima. One way to avoid premature search termination and increase the chance of converging to more fit solutions is to introduce stochastic component into the search process.

Stochastic search algorithms have been already long under focus of ILP research as they allow reaching good concepts in shorter time. Considering the complexity of the search, this feature remains interesting despite that it trades the completeness of the search in favour for fast learning. Different basic randomized search algorithms have been developed and studied e.g. in works of (Zelezny et al. 2006) and (Rückert et al. 2002). In the work (Zelezny et al. 2006) stochastic search for bottom-up rule learning is compared to basic *GSAT* and *WalkSAT* (Kautz et al. 1996), local search algorithms that were primarily developed for solving satisfiability problems. The last implementation of *Aleph* algorithm offers also an adaptation of these standard randomized methods: *GSAT*, *WalkSAT*, *RRR* (Rapid Randomized Restarts). It includes also the Metropolis algorithm which is a special case of *Simulated Annealing* with a fixed '*temperature*' parameter.

The function of GSAT and WalkSAT algorithms in ILP is close to their original role. An ordinary option of GSAT in ILP first draws a random definite clause and instead of changing the logical assignments of individual variables like GSAT it adds or deletes literals in the clause. The ILP variant of WalkSAT is similar, the difference being that with some probability the algorithm makes additional random move (it randomly selects specific literal to add or delete). The specificity of the addition means that the literal, when added, will not cause the clause to cover some negative example; the deletion, on the other hand, will make the clause cover some previously-uncovered positive examples. The experiments show that both algorithms perform better when compared to standard greedy search and are able to find useful clauses in cases where using complete search is intractable.

The *FOIL* system also offers several recent extensions that offer stochastic search. A frequently used extension called *mFoil* is a variant of *FOIL* employing beam search and different search heuristics. Slightly different class of stochastic ILP algorithms is represented by approaches that combine statistical learning with inductive logic programming techniques. The latest of these is the algorithm *nFOIL* (Landwehr et al. 2007), an enhancement of *FOIL* algorithm with Naïve Bayes and probabilistic learning.

An interesting group of randomized algorithms that show good results in solving the search complexity are the nature-inspired techniques. The most known representative of this field is the group of Evolutionary Algorithms (e.g. genetic algorithm). However, the adaptation of these algorithms to ILP tasks is not straightforward and cannot be implemented easily into standard ILP systems. More information about the topic of combining evolutionary techniques with ILP is given in the next chapter.

EAs are not the only stochastic nature-inspired optimization technique that can be used in RL. Another option is represented by Simulated Annealing (SA) (Serrurier et al. 2004). Inspired by physical annealing of solids, it represents an often used alternative to evolutionary techniques. First complete utilization of SA in ILP may be seen in (Serrurier et al. 2004) (although there are some inclinations already e.g. in the Aleph system). This SA adaptation works with ILP clauses and defines their neighborhood by means of generalization and specialization operators.

3.7 Summary

The aim of this chapter was to provide the reader with the basic definitions of ILP and in particular to provide the notions on the standard approaches to solving ILP problems. The advantages of ILP include flexible and understandable representation of relational data and models in the form of clauses in first-order logic and ease of incorporation of background domain knowledge. On the contrary, ILP requires high computational costs compared to standard attribute-value techniques that slightly disqualifies it for standard problems. Also appropriate coding of domain knowledge is not always easy.

ILP can be considered as an optimization problem that is solved through search in a hypothesis space, where the structures are represented in FOL. The objective criterion is to find a hypothesis that covers as much of the positive examples as possible while not covering any the negative ones. The search within the space of FOL clauses is structured as partial ordering by θ-subsumption can be imposed on this space which orders clauses by their generality and allows large parts of the search space to be pruned due to the anti-monotonicity constraint.

We have described three basic ILP systems (FOIL, Progol and Aleph) and also introduced basic extensions of these systems that enhance their function by basic stochastic search approaches. More sophisticated approaches including Evolutionary Algorithms (EAs) usually are not part of these systems and have an implementation of their own. These EA-based systems are described in detail in the next chapter.

4 Evolutionary Algorithms

The aim of this chapter is to introduce Evolutionary Algorithms (EAs) and their implementations in the ILP domain. EAs represent a class of optimization algorithms that operate on the basis of Darwinian evolution principles and utilize these to optimize populations of candidate solutions. After introducing EAs and showing them in a broader context we will give basic flow of the algorithm and describe the basic EA algorithms. In the second half of the chapter we will provide overview of the latest implementations of EAs in the field of ILP and describe several EA-based ILP systems. Finally we will conclude the chapter with comparison of the basic principles employed by these systems and discuss how they relate to tour work.

4.1 Introduction and Motivation

There are many complex optimization problems that currently cannot be solved by exact optimization algorithms. For these hard cases, stochastic optimization methods that employ meta-heuristics can be advantageously used. Although not granting that optimal solution will be found, in most cases these are able to find solution that is close to optimum and often also sufficient enough to be used within the respective application.

Evolutionary algorithms (EAs) represent one class of such stochastic optimization methods that is nature-inspired. Figure 4.1 outlines the position of EAs within the context of all main well-known search and optimization techniques (both deterministic and non-deterministic) (Goldberg 1989). Classical EAs, including genetic algorithms (GAs), evolution strategy (ES), evolutionary programming (EP), and genetic programming (GP) are all space searching population based meta-heuristic algorithms that use random factor when searching for the optimal solution. With these nature-inspired search methods it is possible to overcome some limitations of traditional optimization methods, and to increase the number of solvable problems. Application of EAs to many optimization problems in organizations resulted in good performance and high quality solutions. By using EAs it is possible to solve large instances of problems for which exact approaches fail to find solution in reasonable time.

In the way they work EAs imitate basic principles of life – for each problem they optimize a population of individuals – candidate solutions. For each individual they use the same (user-) defined representation that usually takes form of a set of partial solutions representing one complete solution to the given problem. Complete solution means a complete set of optimized problem control parameters: e.g. a complete schedule in the case of scheduling problems or a complete tour for the traveling salesman problem etc. Influenced by the quality (fitness) of the individuals they reproduce over several generations to create new solutions and search the solution space. Those individuals that are most fit are most likely to survive and are selected to produce their offspring. Fitness of an individual is measured by the evaluation function (or fitness function) that is defined by the problem. Each offspring is created by stochastic application of variation operators to the selected (parent) individuals. By this way the individuals of the population, coupled with the selection and variation operators, combine so that in effect the system performs an efficient versatile domain-independent search.

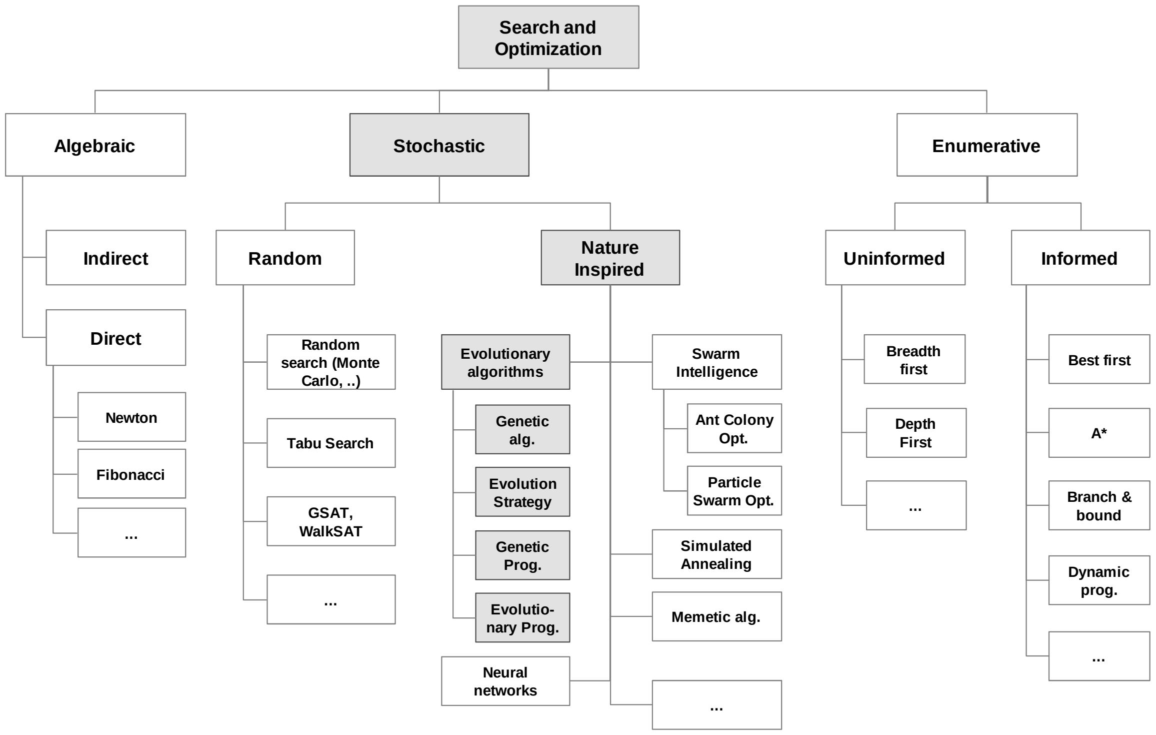


Figure 4.1 – Position of EAs in possible taxonomization of current search and optimization techniques

EAs have been found to be efficient solvers for many optimization problems, see, e.g., the Proceedings of the conferences *GECCO*, *EvoCop*, *PPSN* and other. Large quantity of successful applications to a wide portfolio of problems shows that EAs represent an efficient method for solving complex problems. They were used in many applications solving real-world issues from the basic ones like e.g. sorting and shortest paths or spanning trees through trip and shipment routing to conformational analysis of DNA in bioinformatics, job shop scheduling or engineering applications in which EAs optimize the structural and operational design of machines or factories and many other tasks (Karr and Freeman 1999 or Sanchez et al. 2012). Many of these problems fall into the category of NP-hard problems or even contain more than one such problem and EAs are reported to provide optimal or close-to-optimal solutions in an elegant and efficient way.

The ability of EAs to efficiently solve extremely difficult problems opened the motivation for us to use the EAs as the search algorithm also in the domain of Relational Learning (namely ILP) where the search is NP-hard. Traditionally, ILP is solved by enumerative techniques (complete search) or by greedy search approaches. Recently, several applications were developed that use EAs also for ILP and these are described later in this chapter. However, the application of EAs to ILP is difficult and these systems still show several limitations.

Generally, the four most important aspects of successfully using the evolutionary algorithms are (Goldberg, 1989):

1) *genetic representation* that codes the problem in a form which can be properly varied by the genetic operators (usually the form is an array of codons called chromosome); the encoding used to represent individual solutions strongly affects the behavior of the search;
2) *genetic operators* that generate a new population from the existing population (namely selection operator for solution recombination, variation operators of crossover and mutation and survivor selection operator for population replacement);
3) *evaluation function* (also called objective or fitness function) - the function $f : \{I\} \rightarrow R$, where $\{I\}$ is the set of individual problem solutions coded in selected representation and $\boldsymbol{R}$ is real number; the function represents a heuristic estimation of solution quality that the algorithm uses to drive the search (performed by the variation and the selection operators);
4) *proper set-up of the algorithm parameters* that control the variation and selection phases (these parameters usually include the population size, probabilities of applying genetic operators and number of algorithm iterations).

Considering a broader perspective, prior to accessing to the mentioned four steps, each user needs to answer two main important (and interconnected) questions when implementing EA:

– *What is the proper problem representation that would allow for efficient EA search?*
– *Should the design focus on utilizing operators that are "standard" (general purpose) or develop problem specific operators?*

The crucial factors are to find a proper representation for the problem and to develop appropriate search operators that match well with the properties of the representation. The representation must at least be able to encode all possible solutions of an optimization problem, and genetic operators such as recombination (e.g. crossover) and mutation should be applicable to it. Typically, EAs reach better performance when using operators that are designed to fit the problem characteristics.

Having properly solved the issues above allows the designer to make use of the many advantages of EAs like the robustness, efficient search in large spaces and under strong noise and many others. However, except the advantages we should be also aware of the disadvantages of EAs. The fact that there is no *guarantee* that a genetic algorithm will find a global optimum was already mentioned. The stochastic behavior of EAs is the reason for the incompleteness of the search and often makes EAs to be perceived to operate as "*black boxes*".

In the next section we will give an overview of the basic function of EAs. The domain of EAs became large and heterogeneous and its description is outside the scope of this thesis. In this work we expect that its basics are well known to the reader. Therefore in the following we will just give short description and brief examples of EA algorithms and will rather focus on using EAs in ILP. For more details on EAs see e.g. (Goldberg, 1989), (Koza, 1992) or (Michalewicz, 1992).

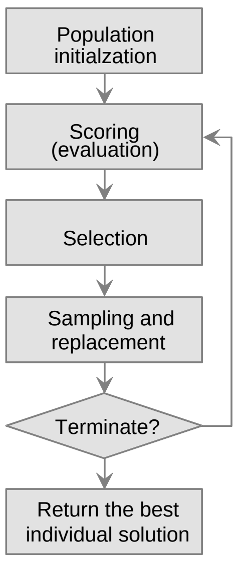


Figure 4.2 - The basic flow of evolutionary algorithms

4.2 Basic Flow of EA

The basic flow that is shared by vast majority of EAs is depicted on the flowchart in Figure 4.2. The flow consists of population breeding iterations within which the competitive selection and random variation takes place. Each iteration of the EA (= each generation) takes incoming set of potential solutions (also denoted as generation) and uses stochastic operations to modify them and produce new individuals. In this view the original individuals play parental role to the newly created modified solutions (offsprings). The offsprings are evaluated and either merged with the original population or they create new population for the next iteration.

The applied selection pressure (selection that prefers individuals with better value of the evaluation function) at the end leads to the result that only the highest fit individuals (better solutions) survive and combine over multiple generations. The process continues until either the limit of maximum iterations is reached or some acceptable solution has been discovered.

The four basic steps of EA have following functions:

1) **Initialization.** In the first step of the algorithm the initialization of the population takes place. This generation of the initial population can be either fully random or some heuristic seeding might take place (in order to focus the search from the starting point).
2) **Evaluation.** The role of the evaluation step is to quantify the quality of each individual in the population. For this purpose, the algorithm uses problem-specific evaluation function that is usually defined by the user. Evaluation serves as the main link between the problem and the optimization algorithm. Values of the function are used to implement the nature-inspired selection pressure. Individuals that represent better solution to the problem at hand have better chances for survival and reproduction. Therefore, the better the evaluation function characterizes the problem the more efficiently EA solves it.
3) **Selection scheme.** Selection is the operator by which EAs mimic the survival-of-the-fittest (natural selection) in the Darwinian evolution. Quality-based selection of parents performed before the cross-over helps to generate good offspring for the next generation. The selection is stochastic yet not fully random. The probability of selecting a individual from current population to serve as parent for the offspring is directly proportional not only to the number of individual parent appearances in the current population but also to the quality (evaluation) of each chromosome.
4) **Sampling.** The purpose of the variation operation is to alternate the selected individuals and on their basis form a new candidate solution. Two operations are usually used within variation step: recombination and mutation:
    - *Recombination* should enable the partial solutions from two parents to be extracted and then reassembled in different combinations. The basic assumption is that as the parents that enter the recombination were selected with regards to their quality and, therefore, the components they are made of should have good quality as well (at least some of them). Recombination of such parents should produce an offspring that has the good characteristics of both of them. Standard recombination operators are 2-parent single- or multiple-point crossovers that split parent chromosomes to two parts (single-point cross-over) or more (e.g. 2-point cross-over or uniform crossover) and reassemble the offsprings from these parts. Apart from these operators splitting the chromosomes at random (blind operators) specialized operators can be used that use more information on localization of the splitting points.
    - *Mutation* is used to randomly alter single individuals. The main motivation for including mutation into the algorithm is to introduce and preserve diversity among the individuals so that the population does not converge prematurely. Based on the situation, the operator can play a role of “background operator” changing the selected

individual only slightly or it can be also used as the main sampling operator to "drive the search" (e.g. when no crossover operators are used).

The simplest representation of individuals that allows to easily use simple crossover and mutation operators is linear string representation. Of course, many other representations exist (often used in so-called hybrid algorithms). An example of a representation that is frequently used is the tree representation where the individuals take form of trees used in Genetic Programming (Koza, 1975). Trees are used in problems where more complex structures need to be used to represent candidate solutions, e.g. mathematical functions as in (Cramer, 1985).
The effect of the random element in the search on one hand allows EAs to solve difficult problems but on the other hand means that the solutions that are found using evolutionary search methods are often not optimal. However, finding sufficiently good solutions is still important for many applications. Additionally, often the solutions discovered by EAs are very close to the optima.

4.3 Basic EA Search Algorithms

Several types of EAs have been developed in the past decades that differ in the representation and the evolutionary operators. Historically we can distinguish four main groups: Genetic Algorithms (Holland, 1975; Goldberg, 1989; Mitchell, 1996), Evolution Strategies (Rechenberg 1998; Schwefel 1977), Evolutionary Programming (Fogel et al. 1966) and Genetic Programming (Koza, 1992). Currently, the most widely used variant are the genetic algorithms.

Algorithm 4.1 - Standard Genetic Algorithm

```
Input: basic parameters (initial population size, probabilities of variation and
selection) evaluation function, termination condition (e.g. number of iterations)
Output: best individual solution discovered

function RunSGA
    INITIALIZE population with random candidate solutions;
    Repeat
        EVALUATE each candidate;
        SELECT parents;
        RECOMBINE pairs of parents (create children);
        MUTATE the resulting children;
        EVALUATE children;
        SELECT individuals for the next generation
    until TERMINATION-CONDITION is satisfied
    end
```

In the following we will briefly introduce following three nature-inspired optimization algorithms: Genetic Algorithm (GA), Evolution Strategies (ES) and Simulated Annealing (SA)[1] – the approaches we tested in our work. The description of the algorithms is short as the field is well known. However, if the reader would be interested in obtaining more information the best sources to start are the above mentioned publications.

4.3.1 Genetic Algorithms

The group of Genetic Algorithms (GAs) represents one of the first groups of evolutionary techniques. The sequence of main algorithm steps is more or less similar to the general flow of EAs (see Algorithm 4.1). In accordance with EAs, the basic steps of genetic algorithms are initialization, evaluation, selection, recombination and mutation. The function of each step was already described in the general section on EAs.

Individuals in GAs are usually represented as linear strings (chromosomes). This is the simplest representation that allows to easily use simple crossover and mutation operators. Many other representations exist (often used in so-called hybrid algorithms).

4.3.2 Evolution Strategies

Apart from the standard genetic algorithm, a variant of mutation-based Evolution Strategies (ES) was tested in this work. Evolution Strategies (Schwefel 1977) is a global optimization algorithm and is one of instances of an Evolutionary Algorithm. The algorithm is similar to the Genetic Algorithm with the difference that they parameterize also the size of the searched neighborhood. The population of the new individuals is created solely by means of the mutation operator that is applied on the selected best μ solutions.

Instances of ES algorithms may be concisely described with a custom terminology in the form *(μ, λ)-ES*, where μ is number of candidate solutions in the parent generation G_P from which new solutions are generated and λ is the number of candidate solutions generated from them that replace G_P in the next iteration. ES defined this way is called "comma-selection Evolution Strategy" in literature.

In addition, a plus-selection variation also exists that is defined in the form of *$(\mu+\lambda)$-ES*. In this alternative the best members of the union of the μ and λ generations compete based on objective fitness for a position in the next generation. The parameters of the algorithm are usually configured so that $1 \leq \mu \leq \lambda < \infty$. The simplest configuration is the *(1+1)-ES* which is a type of greedy hill climbing algorithm.

In this work, the configuration of *$(1, \lambda)$-ES* was tested within the POEMS algorithm. The algorithm maintains one single actual prototype (individual) from which, in each iteration (generation), λ new prototypes are created by hypermutations. If the best individual out of the λ new individuals is at least as good as the parent one then it becomes a parent for the next generation and the current prototype

[1] Although SA do not belong to the group of evolutionary techniques, it is one of the first stochastic nature inspired algorithms that was developed.

clause is discarded. Otherwise, the current parent remains for the next generation. The pseudo-code of the algorithm is given below in Algorithm 4.2.

Algorithm 4.2 The (1, λ)-ES algorithm

```
Input: initial chromosome C, number of offsprings λ
   Iteration = 0
   repeat
      for(i=0;i< λ;i++)
         Offsprings[i] = mutate(C)
      BestOffspring = selectBest(Offsprings)
      If betterOrEqual(fitness(BestOffspring) ≥ fitness(C))
            C = BestOffspring
      Iteration++
   until(terminationConditionMet(Iteration, C))
 return C
```

4.3.3 Simulated Annealing

Although the Simulated Annealing (SA) algorithm (Cerny, 1985) does not belong directly to the class of EAs, it is considered to be the first stochastic nature-inspired optimization algorithm that was developed. SA is mathematical analogy to a process of system cooling which can be used to optimization of highly nonlinear, multidimensional problems. Originally, it was developed as an adaptation of the Metropolis-Hastings algorithm (Metropolis et al. 1953), a Monte-Carlo-based method to generate sample states of a thermodynamic system, developed in 1953. When compared with other EA approaches, SA is simpler to set-up and use but this is often at the cost of substantially long run-times.

In its principle, SA is basically a stochastic hill-climbing algorithm inspired by the process of physical annealing in which liquid systems freeze or metals re-crystallize. The systems are first heated until thermal stresses were released and then cool down to a state with lower energy. The final state can be interpreted as an energy state (crystalline potential energy) which is lowest if a perfectly crystal emerged. The cooling phase to reach the ambient temperature is performed very slowly so that perfect crystals can emerge (the quality of the results strongly depends on the temperature cooling decrements). In SA, the current state of the thermodynamic system is analogous to the current solution to the combinatorial problem, the energy equation for the thermodynamic system is analogous to at the objective function, and ground state is analogous to the global minimum.

By analogy with the physical process of annealing, each step of the SA algorithm replaces the current solution by a random "nearby" solution, accepted with a probability that depends on the difference between the corresponding two evaluation function values and on a global parameter T (called the temperature), that is gradually decreased during the process. The dependency is such that the current solution changes almost randomly when T is large, but increasingly only the "*downhill*" changes are preferred as T goes to zero. The “*uphill*” changes (regarding its evaluation function) are given lower and lower probability of acceptance as T decreases. Although the “uphill” moves actually mean worsening of the solution it is this allowance for "*uphill*" moves that helps the method from becoming stuck at local optima, the bane of standard greedy methods.

Algorithm 4.3 - The Simulated Annealing algorithm

```
Input: temperature T, cooling schedule Alpha, initial chromosome C
      while(not_converged()) do
            C_Adaptation = mutate(C)
            Delta_F = fitness(C_Adaptation)- fitness(C)
            if(Delta_F < 0)
                  C = C_Adaptation
            else
              if(random_number() < e^(-Delta_F/T)
                  C = C_Adaptation
            T = Alpha*T
      end
 return C
```

The pseudo-code of the algorithm is given in Algorithm 4.3. At each step, the SA heuristic considers several randomly created neighbors of the current state *s*. Then it probabilistically decides between switching to the new sequence or staying with the current one. The system usually always switches to state with better fitness while the probability of adopting solution with worse fitness depends on current temperature of the system (and as such is lowering exponentially with time). This mechanism ensures that the system ultimately tends to move to states of better fitness (lower energy). This step is repeated until the system reaches a state that is "*good enough*" for the application, or until termination condition is met (e.g. the computation budget given has been exhausted).

4.4 Softcomputing in ILP

As already stated, the EAs represent a nature-inspired group of randomized search algorithms that is especially suitable for hard problems. These algorithms have been already successfully used for various concept learning problems and it seems that an implementation of EAs into ILP should be logical step to face the complexity of theory induction process (including various systems for ILP (Divina 2006).
When considering the basic four-step structure of the evolutionary algorithms (initialize-evaluate-select-vary&replace) we can see that there are two basic possibilities of adjusting EAs to a new problem. The first involves changing the fundamental GA operators used in the breeding steps (*vary* and sometimes also *select*) to work effectively with often complex non-string objects. The second needs an effort from the authors to develop a mapping of complex concept descriptions into string- or tree-like representation while minimizing any changes to the operators. Both approaches are viable, the advantage of the second approach is that a standard (and even some pre-optimized) GA can be used if an effective translation procedure can be defined. Such mapping must be time efficient and allow the traditional GA operators to manipulate the strings while preserving the syntax and semantics of the underlying concept descriptions. However, as each representation introduces a different solution

neighborhood definition, it usually anyhow requires specific adjustments and implementation of the search variation operators. Additionally, in both cases the efficiency of the system must be supported by efficient individual evaluation procedure. The main optimization criterion is usually defined in advance – accuracy of the final model.
Advantages and disadvantages of each algorithm are often given already by the problem representation it uses. When selecting the proper representation the author must first define what the individuals will stand for in his algorithm. Since the ILP models usually consist of more than one clause, the author must select how EAs will be used to construct the desired sets of clauses. There are currently two basic strategies: the *Michigan approach* (Holland, 1986) and the *Pittsburgh approach* (Smith, 1983) named after the home universities of the research groups that introduced the approaches for the first time. Using the *Michigan approach* means constructing the model in an incremental way – in each step the systems adds another clause to the final model. On the other hand, in the *Pittsburgh approach* each single individual represents the whole model – whole solution to the problem at hand and there is no need for running the system incrementally. Representation is also usually used as one part of solution to the validity issue of EA ILP searchers and background knowledge issue.
Unfortunately, it is not only the representation that is the bottleneck of the use of EAs in ILP. The implementation cannot be straightforward by simply using evaluation function standard for ILP (e.g. model consistency – see Chapter 3) and adjusting the EA search operators. This is mainly due to the fact, that the ILP problems are extremely complex: the corresponding searched spaces of logical hypotheses defined in FOL are very large and also evaluation of each candidate solution (logical clause or model) takes considerable time. Following main issues/challenges currently still hinder wider application of conventional evolutionary algorithms in ILP, last three identified already in (Reiser 1999):

1) **Efficient problem representation.** All current EA-based ILP searchers use specialized representation of the clauses to be able to translate them to a form that EA systems can operate with. These representations do not often allow to fully use all capabilities of ILP refinements and additionally require development of specialized search operators. Eventually, EAs for ILP often become hybrid systems that are highly specialized on solved ILP tasks. Any transfer to different problem requires considerable user inputs which actually stands in contrast to one of the basic characteristics of EAs – the domain independency.
2) **Efficient concept evaluation.** Clause coverage calculations (used to determine candidate concept quality) represent NP-hard problem and as such are time consuming – this is important issue for EAs because they are based on working with populations of potential solutions which brings correspondingly large number of evaluations.
3) **Efficient search for reasonable concepts.** The standard ‘blind’ evolutionary operators construct mainly candidate concepts that are either too general (cover all given training examples) or are meaningless in the problem domain (semantically invalid, do not cover any example); this issue is additionally amplified by specific characteristics of ILP domains where the presence of meaningful concepts among the rest of theoretically possible ones is sparse;
4) **Refining discovered concepts.** Despite the fact that EAs perform well at locating good solutions in complex search spaces, they are poor at refining these solutions;

5) **Domain knowledge utilization.** Generally, EAs perform well in the absence of problem domain knowledge (DK), but when such knowledge is available, it is difficult for EAs to exploit it either in the representation or in the evolutionary search. According to (Tamaddoni-Nezhad, 2002), EAs and ILP are on opposite sides in the classification of learning processes. While EAs are known as empirical or DK-poor, ILP could be considered as DK-intensive method in this classification.

Due to these five issues, the incorporation of EAs into ILP requires special treatment and often results in "*hybrid*" solutions. Various approaches have been developed so far. For example, several evolutionary ILP concept learners take the simplest approach where the EAs just replace the commonly used hill climbing method for finding the best rule to be added into the emerging target concept like e.g. the system REGAL (Giordana et al. 1996). Another example of hybrid approach introduced by hybrid system EVIL1 (Reiser 1999) uses the ILP refinement algorithm Progol to evolve population of logic programs (individuals). The EA feature utilized is the recombination operators used to swap parts of concepts between individuals selected from this population.

Of course, not all EA-based ILP systems are this simple. Besides the simple greedy rule-by-rule construction strategy there are also more global search strategies, which make better use of the population-based evolutionary search. In these systems, the EA is iteratively used to evolve a population of candidate rules, from which a number of the best ones is added to a pool of promising rules. At the end, after all of the positive examples have been covered the target concept is extracted from the pool of rules. Examples of the latter approach are the Evolutionary Concept Learner (ECL, Divina 2006), G-NET (Anglano et al. 1998) or DOGMA (Hekanaho 1998). These systems use a concept of parallel evolutionary models and a co-evolutionary strategy, respectively, to search the space of whole target concepts.

In the following we give a brief overview of the most known systems. We can see that a number of EA-ILP systems has been developed coming with various ideas of how to use the EAs on the field of ILP. Looking at the portfolio of algorithms through the structure given above we will briefly look at the solutions they adopt mainly regarding problem representation, search operators, search methods and evaluation functions used. After the description we will continue with comparison of the existing algorithms and discussion.

4.4.1 REGAL

REGAL (Neri and Saitta, 1995; Giordana and Neri, 1996) is a FOL concept learning algorithm. REGAL stands for RElational Genetic Algorithm Learner. It can be considered as the first extension-driven method to stochastic search in the field of machine learning. REGAL operates with a distributed genetic algorithm-based system, which is created for learning FOL concept description from given examples.

Representation. The algorithm uses simple fixed-length binary string representation. However, compared to standard ILP system, REGAL operates with a much simpler language than full FOL. It restricts the hypothesis space by means of so called "Language Template Λ". Clasues Λ of language template Λ have to include at minimum one predicate in completed form. The selected language template limits the search space, which is studied by REGAL, to the set $H(\Lambda)$ of formulas, which can be reached by excluding some constants from the completed terms occurring in Λ. The admissible conjunctive concept descriptions can be obtained from Λ.

Search operators. REGAL tries to limit the search space and number of irrelevant clauses produced by adjusting the selection mechanism. Individuals are selected for reproduction by means of the Universal Suffrage selection mechanism (Giordana and Neri, 1996). This selection mechanism actually combines informed selection with standard roulette wheel selection. First, a sub-group of individuals is selected from the whole population so that the group contains only individuals covering the same examples. After that, fitness proportional roulette wheel selection is used to pick the parents for the next offspring.

Evaluation. REGAL creates a mapping between logical expressions and fixed-length binary strings. To evaluate strings they are mapped back to the corresponding logical expression which may then be interpreted. REGAL utilizes the set covering algorithm as well as e.g. the FOIL algorithm. The evaluation is performed each time on the full training set.

Search algorithm. The system operates with a network of genetic nodes fully connected with each other and exchanging individuals at each generation. Each individual possesses one single clause (i.e. a partial solution of the overall theory). Each individual originates separately and just, when the run ends, a complete theory is created. The creation of the partial solutions is guided by supervisor node. The supervisor node manages the periodically extracting a classification theory from the nodal populations and changing the focus of the hypothesis space exploration any time when a significant change in the classification theory is detected. When no further improvement is recorded, the description is saved in order to release the covered positive examples from learning set.

4.4.2 G-NET

G-NET stands for Genetic NETwork (Anglano and Botta, 2002). It is an inductive learner, which is based on Genetic Algorithms (GAs), created to reach disjunctive classification programs described in FOL from structured pre-classified data. It is a successor of REGAL algorithm but differs in several ways. G-NET was originally structured as a distributed system with a collection of separately running processes that use genetic algorithms (genetic nodes) and one supervising process that manages the search and merges the partial results from genetic nodes to the final outcome.

Representation and search operators. G-NET is operating with the same bit string representation language adopted by REGAL. Two-point crossover operator is used for recombination while blind mutation operator is replaced by generalization/specialization operator.

Evaluation. G-NET works with two evaluation functions that again differ from the REGAL approach. The first one works at a global level assessing quality of created clause by measuring quality of the whole theory. The second function works at local level and evaluates quality of clauses in each genetic node. Another difference between REGAL and GNET can be seen in the selection operator. G-NET is not working with the universal suffrage operator but, instead, individuals are chosen directly according to the fitness proportional selection only.

Search algorithm. G-NET uses a co-evolution strategy by means of two algorithms. The first algorithm focuses on the global concept description combining results of the most fitting hypotheses emerged in various genetic nodes. The second algorithm counts the assignment of the positive concept instances to the genetic nodes. The strategy focuses the search on the concept instances, which are covered by poor hypotheses, without bypassing to continue the refinement of the other hypotheses.

4.4.3 DOGMA

The DOGMA (acronym stands for Domain Oriented Genetic Machine) system operates under two distinct levels and it uses two separate languages for describing the examples and the hypotheses (Hekanaho, 1998). The advantage of this two-level approach is that extension of a given hypothesis with respect to the example set can be realized by logical deduction scheme that is comparable to resolution. To incorporate background knowledge into the learning process DOGMA uses the operator of background seeding. The principle is to create new chromosomes by randomly selecting parts of background rules and translating these parts into chromosomes.

Representation. DOGMA uses the same type of representation language and encoding as REGAL – mapping of individual clauses that have to be in accordance with given language template, to bit strings.

Search operators. There are 5 genetic operators used in the algorithm. Apart from mutation and crossover operators (taken from REGAL), mating, seeding and background seeding operators are used. The mate operator is a recombination operator designed for the species-based approach that DOGMA uses. Based on user-defined sensitivity parameters, it mates two selected chromosomes sensitive to the species of the respective chromosomes. The seed operators were designed to cope with the specific requirements of ILP. It generates a formula (bit string) that covers at least one randomly selected positive training example. Background seeding is almost the same as seeding but it generates bit strings by selecting random parts from a selected background rule. Additionally, DOGMA uses several remaining operators work on the higher level (family level).

Evaluation. Like the whole algorithm, also the evaluation step is split into two levels and the evaluation function is operating with two different functions. The first (lower level) is based on the MDL principle, and the second is based on the information gain measure.

Search algorithm. On the lower level, the algorithm is constructing the model in an incremental way – in each step the systems adds another clause to the final model. The higher level on the other hand operates with the whole model at once. Fixed length bit string chromosome representation is used on the lowest level of the system. Lengths of the chromosomes are guided by crossover and mutation operators. On the higher level chromosomes are put together into genetic families, through special search operators which are able to merge and break families of individuals.

4.4.4 GLPS

In contrast to previous approaches that use bit string representation, the Genetic Logic Programming System (GLPS, Wong and Leung, 1995) uses a tree-like representation in which each individual takes form of a forest that describes the theory. However, authors of the system also concluded that the standard Genetic Programming (GP) equipment is not sufficient to solve the ILP learning problem and, as a consequence, they developed new specialized operators. In effect, the GLPS system searches the space only by means of operators similar to basic GP crossover. However, several other limitations led the authors to stop development of GLPS and to start another EA-ILP system this time using more structured representation – grammars. This led to creation of the LOGENPRO system (see below).

4.4.5 LOGENPRO

LOGENPRO (the LOgic grammar based GENetic PROgramming system) is a system that combines GP and ILP through a formalism of logic grammars that are used to specify the search space declaratively (Wong 1998, Wong and Leung 2000). Being an extension of the GLPS (Wong and Leung 1995), the system utilizes the context-sensitive information and domain-dependent knowledge used to accelerate the learning of knowledge. For this purpose logic grammars are employed to provide declarative descriptions of the valid programs that can appear in the initial population.

Representation. Common ML systems that use grammars within the learning process, e.g. Grammatical Evolution (O'Neill and Ryan, 2003), use context-free grammar in the notation of BNF (Backus-Naur form). In contrast to this LOGENPRO uses logic grammars (see example in Table 4.1) because context-free grammars are not expressive enough to represent context-sensitive information of the language and domain-dependent knowledge of the target program being induced. The difference lies in the fact that the logic grammar symbols (both terminal and non-terminal) may include arguments, i.e. logic variables, functions or constants (constant is simply a 0-arity function). This enables to use arguments in the grammar and thus to enforce context-dependency.

Table 4.1 – An example of a part of simple logic grammar used by LOGENPRO (Wong and Leung, 2000)

```
R#1: start -> {member(?x,[X, Y])}, [(/], exp-1(?x), exp-1(?x), [)].
R#2: exp(?x) -> [(+ ?x 0)].
R#3: exp-1(?x) -> {random(0,1,?y)}, [(+ ?x ?y)].
...
```

Evaluation. The authors used standard fitness function based on the number of misclassified examples in the selected training set.

Search operators and search algorithm. LOGENPRO solves the tasks of inducing the logic programs by means of search for a highly fit program in the space of all possible programs in the language specified by the logic grammar. LOGENPRO starts with an initial population of programs generated randomly, induced by other learning systems such as FOIL (Quinlan 1990), or provided by the user. During the search, grammars are used to disallow crossovers between individuals (represented as derivation trees) or mutation that would lead to creation of syntactically incorrect individuals.

4.4.6 SIA01

SIA01 (Supervised Inductive Algorithm version 01, Reiser and Riddle, 2001) works with a sequential covering principle proposed in AQ (Fuernkranz, 1999) and uses a bottom-up approach when inducing individual clauses.

Obj	*X*	*Color*	*X*	*blue*	*shape*	*X*	*square*	*far*	*X*	*Y*	*2*

Figure 4.3 – Encoding of individuals in *SIA01*, representation of clause:
`obj(X), color(X, blue), shape(X, square), far(X, Y, 2)`.

Representation. The scheme used in the GA for representation of individuals is given in Figure 4.3. It shows the encoding of the clause '`obj(X), color(X, blue), shape(X, square), far(X, Y, 2)`'. The linear array scheme implemented in SIA01 differs from REGAL as instead of adopting a bit string representation for encoding clauses, SIA01 uses a higher level encoding. FOL notation is directly used - predicates and their arguments are treated as individual genes. This representation gives more flexibility than other representations used in EA-based ILP systems as it does not fix the length of a chromosome neither it requires the user to specify some initial template for the target predicate. Additionally, the higher level representation allows using directly the FOL refinement operators like clause generalization or specialization.

Search operators. SIA01 uses common genetic operators adapted in following way. The crossover operator is restricted to one-point crossover with the cross point having to lie in front of a predicate. Each offspring after restricting one-point crossover must be checked on semantics and to be confirmed to as being legal. For mutation, four generalization mutation operators and four specialization operators are used with the probability of application depending on fitness. When the fitness value of an individual is lower than a threshold value, the application probability of generalization operators is larger than the application probability of specialization operators.

Evaluation. The algorithm uses specific compound fitness function that combines four elements in a weighted average. The elements used are the coverage, the number of literals, number of variables and uniqueness of the individual within the population.

Search algorithm. Before the search, *SIA01* randomly picks an uncovered positive example and applies it as a seed to create the first clause. Thereafter, it searches for the most fitting generalization of this clause optimizing the evaluation criterion. This process runs with utilization of a GA. To acquire new generation the algorithm applies a genetic operator onto each individual in the population inserting the new individuals into the population. The size of the population is able to grow in this way until it reaches a predefined threshold.

4.4.7 EVIL1

EVIL1 (Reiser and Riddle, 1998), that stands for Evolutionary Inductive Logic programming, is a hybrid EA-ILP algorithm that consists of a GA system co-operating with Progol to solve the ILP tasks. This system uses so called Pittsburgh approach - the evolutionary component optimizes the global model and evolves the population of clausal theories (logic programs).

Representation. Every individual represents set of rules (logic program) described by a tree structure. Each node of the tree stands for a single clause. The clause tree is a rooted binary tree where the vertices are either the clauses induced by Progol or null. As a result, the theory is not only represented as a set of clauses but additional information is stored that captures relationships between them. In this approach it is possible to come up with a whole logic program inside a single individual.

Search operators. Mutation operator is not used in EVIL1. Instead, Progol ILP algorithm is used to evolve selected candidates. The crossover operator is similar to crossover in Genetic Programming – it randomly exchanges sub-trees between the selected two parents. Special mechanism is proposed so that the clause trees should be disrupted as little as possible by stochastic evolution.
Evaluation. The fitness of the discovered concepts is defined as the accuracy on a full training set.
Search algorithm. At each generation, selected individuals are refined by an ILP engine Progol inducing new rules. Then, parts of rules are exchanged by crossover operator and, finally, population for next generation is stochastically selected based on validation set accuracy.

4.4.8 ECL

ECL (Evolutionary Concept Learner) introduced in (Divina, 2008) is an ILP system using a selection-mutation search process that can be labeled as a hybrid EA. The approach of the algorithm is closer e.g. to Evolutionary strategies than classical GA.
Representation. ECL uses a high level representation that can be considered similar to representation used by SIA01. Additionally, in order to represent clauses of variable lengths it treats the atoms of the clause either as active or inactive. Atoms that are semantically incorrect are marked as not active and not considered when the clause is evaluated. However, during the evolutionary process these can be activated and used for creating new offsprings.
Search operators. Only mutation operator is used in the system without any recombination operator (e.g. crossover). Four mutation operators are used for the main search in the FOL space - two mutation operators for specializing the clauses (add an atom to the clause body or turn variable into constant) and two mutation operators for generalizing. The mutation process is not random but can be considered more as a greedy-like search. After considering a predefined number of mutation outcomes the best one among the options is applied. Additionally, each mutation is followed by an optimization step. Optimization means repeated greedy application of the operators to the selected individual until its fitness does not increase (or the constraint of maximum number of iterations is met). Also, specialized selection mechanism was developed for the ECL system.
Evaluation. To shorten the evaluations ECL employs random sampling of the background knowledge and uses only partial background knowledge for evaluating its individuals. At each execution of the EA, a part of the background knowledge is chosen by means of a simple stochastic sampling mechanism. Additionally, to increase the classification ability over numerical data the authors complement the algorithm with several unsupervised discretization methods. The fitness function used by the authors is inverse value of accuracy function (inverse ratio of examples classified as positive to number of all examples).
Search algorithm. The ECL system iteratively breeds a population of FOL models as the union of several sub-populations evolved with an EA. Each population is evolved by the repeated application of three phases: selection, mutation and optimization. At each generation, *n* individuals are selected using modified selection operator that enforces the population to reach maximal coverage over the training data (individuals covering the same examples compete with each other for being selected). Each individual then is adjusted in two steps - in mutation phase and in optimization phase. After the adaptation, individuals are either simply inserted into the population or contest for their position in tournament-based selection process.

4.4.9 FOXCS-2

In this section we will briefly present the FOXCS-2 ILP system (Mellor 2008). Although it does not use EA for its work it is related to our work because it uses refinement actions to solve the task in a way that can be considered as similar to the POEMS-based solution introduced in next two chapters of this work. FOXCS-2 is a reinforced-learning-based classifier system that was developed to evolve FOL rules. The system is an extension of the XCS learner so that it learns rules expressed as definite clauses in FOL and therefore is capable to handle the ILP problems. The rule-based approach adopted by the system allows for incrementally learning expressions over first-order logic.

Representation and search operators. The system uses the first-order logic representation and special evolutionary operators - generalizing and specializing mutation operations are used to search the space. Generalization is preformed at random via 3 operators: *delete literal*, *constant to variable* and *variable to anonymous variable*; specialization is performed by three other operators: *add literal*, *variable to constant* and *anonymous variable to variable*.

Evaluation. The system uses accuracy-based fitness to evaluate quality of the rules and give credit to the learning engine. All examples in the training set are considered during evaluation.

Search algorithm. The system generates, evaluates, and evolves a population of rules that have the form "*condition - action*". These rules are represented as definite clauses in FOL. In contrast to other systems that use reinforced learning, the system is general (applies to arbitrary Markov decision processes), model-free (rewards and state transitions are "black box" functions), and "tabula rasa" (no initialization with pre-defined candidate rules required). The input to the algorithm is set of applicable actions by which the system can alter the initial rule. The search is then performed by learning engine that, based on the current status of the rule, stochastically selects from the set of applicable actions such action that alters the rule in the best way. The action selection mechanism offers two options: based on generated random number and user-defined probabilities, either a random or the best rule-modifying action is selected from the action set.

4.5 Comparison and Discussion

In this subsection we want to provide a comparison of the existing EA-ILP approaches in the main dimensions. Let us just briefly recall that these are (re-ordered and grouped so as to be closer to the sequence of their utilization within the algorithm):

1) problem representation and domain knowledge utilization,
2) search for reasonable concepts and refining discovered concepts,
3) concept evaluation.

These three topics are briefly discussed in the following subsections. The approaches of existing EA-based ILP searchers described in the previous section to these points are summarized in these tables - Table 4.2 (type of refinement used and type of EA operators used) and Table 4.3 (representation and evaluation methods used).

Table 4.2 – Summary of EA operators used by the EA-based ILP systems (*gen = generalization, spec = specialization*)

System	Type of refinement Implemented[1]	EA operators	
		Recombination operator	Mutation operator[1]
REGAL	gen + spec	uniform 2-point crossover	classic
G-NET	gen + spec	uniform 2-point crossover	special (gen + spec)
DOGMA	gen + spec	uniform 2-point crossover	classic
GLPS	gen + spec	subtree exchange	none
LOGENPRO	gen + spec	subtree exchange	none
SIA01	gen (realized through 4 mutation operators)	1-point crossover (classic & restrained)	special (gen)
EVIL1	gen+spec	subtree exchange	none
ECL	gen + spec (4 mutation operators)	None	special (gen + spec)
FOXCS-2	gen+spec	None	spec. (gen + spec)

[1] gen = generalization operator, spec = specialization operator

Table 4.3 – Summary of problem representations and evaluation techniques used by the current EA-based ILP systems

System	Problem representation		Approach to evaluation
	Individuals	Model	
REGAL	bit strings (user defined template)	Michigan approach	Full example set eval.
G-NET	bit strings (user defined template)	Michigan	Full example set eval.
DOGMA	bit strings (user defined template)	Michigan & Pittsburgh	Full example set eval.
GLPS	and-or trees	Pittsburgh	Full example set eval.
LOGENPRO	grammar derivation trees	Pittsburgh	Full example set eval.
SIA01	higher level representation	Michigan	Full example set eval.
EVIL1	Clause tree representation	Pittsburgh	Example subset sampling
ECL	higher level representation	Michigan	Example subset sampling
FOXCS-2	higher level representation	Pittsburgh	Example subset sampling

4.5.1 Problem Representation

The definition of the concept description language is important factor of any EA-based algorithm as it introduces a language bias that can promote of disqualify generation of some types of concepts. There are three types of representation of individuals that are used among the EA-based ILP systems: bit-string, tree representation and higher level representation (see Table 4.3 for summary). While the first two can build on standard EA principles and theoretically use slightly modified GA resp. GP operators, the higher level representation requires more specialized search operators.
The systems using *bit string representation* (DOGMA, G-NET or REGAL) are limited to solutions that can be described by user defined template of constant length. Motivated by ease of the variation operators implementation, this solution in fact represents a form of search using propositionalization.

Moreover, the conversion of the ILP problem to propositionalized form is actually performed by the user of the algorithm that supplies the template. This utilization of the templates imposes strong limits not only on the type of achievable solution but also on the search process as the representation does not enable to perform most of the FOL refinement operations, e.g., changing a constant into a variable. Additionally, the user needs to have some knowledge of what the expected solution should look like and be also able to code such template into the algorithm. Further complications may come when dealing with numerical values or more complex datasets where the template string can become too long and may slow down the learning process.

In this approach it is very difficult to make use of background knowledge as it needs to be selected to be a part of the template. On the other hand, user can easily control the language bias and validity of the produced solutions – only those clauses that fit the template can be construct[2]ed.

The second option to representation is to use the *tree structures*. First popularized in EAs in Genetic Programming (Koza, 1992), the trees are representation that offers variability of the concept size, modularity and flexibility needed for describing concepts in FOL. Clauses do not map naturally to fixed linear strings as they are of variable length. As such they can be better represented by a tree. The model is in this case represented as a forest of AND-OR trees, where the inner nodes are logic dis-/ conjunctions and the leaves of the trees contain predicate symbols and terms of the problem domain. Special tree manipulating operators are available in the GP systems. This approach is used by the system GLPS (Wong and Leung 1995) a system that evolves a population of complete tree-coded logic programs. The tree representation of concepts shows to be more flexible as it allows considering rules of variable form. Such representation also allows utilization of standard tree-based crossover operators, where rules, clauses, or just parts of clauses can be easily swapped between two theories.

In the case of trees, validity of individuals can be either ensured by special variation operators or forced out through grammars. Yet, the second approach needs some adjustment of the variation operators as well as it is needed that they adopt the grammar as well. Utilization of background knowledge is less problematic – the predicates need to be inserted into the evolved trees and the local neighborhood of each predicate can be used as source of context information that not only helps creating semantically valid individuals but also enables to use parts of background knowledge on meaningful places.

The *higher level representation* implemented in the SIA01 or ECL systems is more flexible than previous two cases. The form of discovered clauses can vary as the structure of each clause is determined by the positive example selected to be the seed in the initialization phase. This representation allows describing flexibly the ILP problem defined in FOL, however special ad-hoc operators are required to manipulate with the individuals. The grammatical representation used by the system LOGENPRO represents also an interesting option. However, as can be seen also on the given example (see Table 4.1) the logic grammars are complex and not transparent structures that have to be defined by the user. This complicates their utilization.

[2] These advantages and disadvantages mentioned hold also for the latest representation approaches of (Tamaddoni-Nezhad, 2002) that was not mentioned here that uses specially constructed binding matrix to transform the problem into binary array representation.

4.5.2 Search and Refinement Operators

The most popular approach to using EAs in ILP currently is developing new EA search operators with an effort to make them sensitive to the syntax and semantics of selected concept description (see Table 4.2). The standard blind evolutionary operators can construct candidate solutions that either do not represent a valid classifier (syntactically invalid) or are meaningless in the problem domain (semantically invalid). For this purpose, the adjustment of variation operators is an approach to solve the issues of efficient search, refinement as well as background knowledge issue. Restricting the space of potential FOL models and incorporation of background knowledge may be addressed by using constraints like grammars which requires adjustment of both representation and the operators.
Despite this the EAs still need to face the problem of refining the FOL classifiers they construct. This makes the approaches modify the standard recombination-mutation scheme and supplement the basic genetic operators with FOL refinement operators that add or remove literals, constants or variables from the clauses. Usually mutation operator is used as an interface for this purpose – this is case e.g. of the REGAL or DOGMA systems. Additionally, in an effort to improve search performance of EAs in local refinement, EAs are often complemented by a local search method (e.g. ECL). The approach then refines the individual solution and instead of considering the entire search space, a smaller neighborhood is examined.
The simple crossover operator, on the other hand, is the mostly used recombination operator with string or tree representations. An example of this solution is the GLPS system – it does not use any mutation operators and the reproduction phase is carried out only by the crossover operator, which can exchange part of the trees (rules, clauses or literals). On the other hand, in the systems that use higher level representation the crossover operator is left out (e.g. SIA01, ECL) – this is a consequence of the fact that ILP clauses and models usually do not consist of several independent parts where each of which has its quality without respect to the remainder. Changing part of the individual usually affects the quality of the other parts and the whole individual quality deteriorates. As a result, most of the search in higher level representation is performed by mutation operator working in a manner close to hill climbing search. The main factor that uses positive characteristics of EA that were the reason of implementing EAs into ILP is the population-based search. However, limiting search to local mutation must make the systems vulnerable to local optima and plateaus.

4.5.3 Evaluation

Various combinations of completeness, consistency and simplicity of individuals are used in the fitness functions of EA ILP searchers. Generally, we estimate that standard ILP engines like SWI-Prolog are used to perform coverage calculations when evaluating the individuals (Divina, 2006).
Efficiency of the evaluations is not the primary task of current systems. The systems do not make difference between generalization and specialization to use the anti-monotonicity based search space pruning. The systems that utilize the Michigan approach usually employ set-covering to sequentially shrink the size of training examples. An interesting solution to optimization of evaluation time when using Pittsburgh approach is introduced by ECL. It counters the lengthy evaluations by sampling of the background knowledge and example set – only a small subset is used to perform the evaluations.

Apart from classification quality assessment, several systems use the fitness function to bias the search process towards simple classifiers and introduce the simplicity factor (e.g. example the MDL principle used in the systems that use string representation like DOGMA and REGAL). In the systems that use parallelization and co-evolution of sub-populations like G-NET fitness function is also used to promote correct cooperation of the individuals in the final model (G-NET actually uses two fitness functions, one used at a local level, in the genetic nodes, and another one used at a global level).

4.6 Motivation for Further Research in Combining EAs and ILP

From our perspective, EAs represent an interesting method that can efficiently complement current ILP techniques and offer efficient solutions to solving ILP problems. However, considering the described issues, it seems that a lot of adjustments and special implementations are required to use EAs on the field of ILP. Still, the results are not fully in accordance with general expectations we have from a standard ILP system – use fully the capabilities of FOL. Such system should be able to utilize not only background knowledge but also the standard ILP refinement operations (e.g. unify two variables or turn variable into constant) so that the result is not unnecessarily restricted by external bias. Additionally, the problem of utilization of any ILP model search algorithm comes when we consider the sparse space of valid concepts. To efficiently search such space we require that both initialization and the search operators produce (mostly) syntactically and semantically correct individuals.

Namely, we can see possibilities for efficiency improvement of EA-ILP combination mainly in the following three areas:

a) **Problem representation and mapping.** A method of representation that would allow to fully use all possibilities of FOL but would also be simple to allow for efficient implementation would help speeding up the search process. The EAs proved to be useful when they can operate with simple modular representation (e.g. string arrays or tree representation) that enables easy and effective crossovers and mutations. FOL is not an easy challenge – the model can consist of number of clauses that is unknown prior to start of the algorithm and, moreover, the clauses made up by logical conjunctions of various lengths with variables, constants and numbers are difficult to map not only onto linear strings but also to tree structures. Higher level descriptions appear in the literature but these are often difficult to manipulate with and require development of specialized EA operators.

b) **Search and refinement of the concepts.** An efficient approach to implement ILP search operators in a manner that could be used by EAs would help constructing less problem specific and simpler EA-ILP systems. To search the space, EAs use the recombination and mutation operators that search the space "globally" (i.e. not by browsing only through local neighborhood of discovered concepts). Due to the sparse nature of the ILP search space the standard blind crossover and mutation operators must necessarily produce undesirably large amount of syntactically or semantically useless solutions because of indefinite amount of theoretically possible atom combinations. On the other hand, the standard "local" ILP refinement operations are not easily translatable to recombination and mutation operations. Therefore, special operator modifications are necessary and used in all EA ILP systems to restrict the search space to focus the search on producing reasonable concepts. This leads to specific solutions that lead to complex hybrid algorithms and in effect hinder wider utilization of EA in ILP.

c) **Efficient concept evaluation.** Focus on efficient concept evaluation techniques should strongly speed up the search process of EA-ILP systems. The EAs usually produce large amount of candidate that need to be evaluated. Tedious calculation can be one of the factors that flip the coin against selecting the evolutionary approach to solve ILP problem and as such evaluation process should be given strong attention. Most of the learners above use directly the basic Prolog-based solvers to evaluate the clauses. An often accepted optimization solution is to sample the example set into smaller subsets and run the tests separately on the niches (Pittsburgh approach, e.g. G-NET) or use the set-covering method to construct the model (Michigan approach, e.g. SIA01). In the EA-based ILP-solvers, the fitness function is usually used in a standard way just to evaluate individual solutions, it is not usual to employ the fitness function to specially bias the search through towards semantically or syntactically valid classifiers.

We believe that these gaps were still not fully filled by any of introduced systems. Therefore, in this work we introduce a new approach for combining EAs with ILP. In our approach, we do not want to create another problem specific EA-based ILP searcher. According to us, one of the main characteristics of EAs is their domain independency. Ideally, ILP problems would be only one of many problems the algorithm would be able to solve and the system would be able to use as much of the existing standard approaches as possible.

This decision to adopt such universal and modular approach that would use most of standard EA searchers has immediate implications for the choice of problem description language - the only possibility for efficiently realizing this system lies in the representation/mapping area. We thus need to search for such approach that can effectively bridge EAs domain that is built upon using string (or tree-like) representations and FOL domain that describes the concepts in versatile language of first order logic. Additionally, such approach needs to have the necessary feature to allow EAs construct and refine syntactically and semantically correct solutions.

We think that a proper candidate to face these requirements might be found in adapting the general algorithm called ***Prototype Optimization with Evolved Improvement Steps*** – POEMS (Kubalík and Faigl 2006) for the purposes of ILP. In the following two chapters we present description of the algorithm as well as the details on our adaptation of POEMS to ILP problems.

4.7 Summary

In this section we have introduced basic principles of Evolutionary Algorithms and described recent approaches of implementing these algorithms to the field of ILP. Inspired by Darwinian principles of evolution and genetics, the Evolutionary Algorithms represent a group of domain independent optimization algorithms. By means of simple mechanisms, the EAs exhibit complex behavior and ability to solve difficult problems with reasonable computational requirements.

However, several drawbacks complicate the direct utilization of EAs in the field of ILP: EAs have poor local search properties and the domain knowledge cannot be used easily with EAs. These problems need special treatment and are in current EA-based ILP systems solved by specific approaches - development of special representations and special search operators. Thereby the systems become highly specialized on ILP tasks and loose the main advantages of EA systems – simplicity of implementation and maximal domain independency.

We have introduced several solutions to use EAs in ILP. We can see that almost all of the introduced EA-based ILP concept learning approaches do not care about minimizing syntactic errors and involve much effort in formulating a suitable representation and search operators. In addition, this formulation must be usually re-addressed for each problem domain thus making the systems hard-bound to certain problem domain. They also cannot employ the anti-monotonicity pruning and most of the above mentioned EA approaches even struggle with the background knowledge. Generally the domain knowledge is used only for the purposes of generating the initial population (if it is used it at all). As a result, the majority of randomized approaches to ILP fail to focus the search on syntactically correct hypotheses and in consequence these algorithms perform the search in too large space while being restricted to specially defined schemes.

In response to these issues we will use the refinement-based algorithm named Prototype Optimization with Evolved Improvement Steps (POEMS) to solve the ILP problems by means of EAs. The algorithm is described in the next chapter and the description of how we adjusted it and used for ILP follows in the consecutive chapter.

5 POEMS Algorithm

This section gives a description of the design of the "***Prototype Optimization with Evolved iMprovement Steps***" (POEMS) evolutionary-based optimization framework. This is a recently introduced new iterative general optimization framework that is based on utilizing an evolutionary algorithm where the main principle of the search for the optimal solution is the process of incrementally improving some given initial concept solution. The framework showed good performance on hard optimization problems - both large discrete problems and real-valued optimization problems (Kubalik, 2011), (Kubalik et al. 2010). It exhibited much better search capabilities compared to the standard evolutionary approaches especially on discrete optimization problems such as the traveling salesman problem which shows that POEMS can be efficiently used also in ILP. The description in this chapter primarily focuses on the core characteristics of the algorithm with the perspective to use it for solving ILP tasks and assumes that the reader is familiar with basic concepts of evolutionary algorithms.

5.1 Introduction

In the standard evolutionary optimization framework, the evolutionary algorithms (EAs) are typically used to evolve a population of candidate solutions to a given problem. Each of the candidate solutions encodes a complete solution – i.e. the complete set of problem control parameters (e.g. a complete tour for the traveling salesman problem). This implies that especially for large instances of the solved problem the EA searches enormous space of potential solutions. EAs have recently demonstrated very good performance in solving these tasks and this along with their flexibility gives motivation for their utilization also for Relational Learning (RL) problems.

Because of the complexity of RL it seems that an implementation of stochastic optimization algorithms such as Evolutionary Algorithms (EAs) (Goldberg 1989) also into this domain should be logical step towards the optimization of theory induction process. However, the use of EAs in RL is rather complicated. Following three main issues were identified already in (Reiser 1999) that still hinder wider application of conventional evolutionary algorithms in RL:

1) *Focusing the search on meaningful concepts.* The EA search samples too large search space which is a drawback in situations when the occurrence of meaningful concepts in this space is sparse.
2) *Refining discovered concepts.* EAs show to have weaker local search characteristics than other optimization algorithms. However, due to the specificity of RL, incremental refinement search techniques proved to be useful when tackling the search complexity.
3) *Exploiting domain knowledge.* One of the main sources of the EA flexibility is their domain independency. Yet, this is also the reason why it is difficult to use domain knowledge when it is available.

To these restrictions we can add following two requirements that are expected from ILP learners but with which current GA-based systems struggle with:

4) utilizing fully the capabilities of FOL including the refinement operators;
5) using reasonable language bias for representation and the operators that is not too restrictive and not too complex to implement;

According to us, the ***Prototype Optimization with Evolved Improvement Steps*** (POEMS) algorithm (Kubalík and Faigl 2006) is the approach that helps overcoming these basic deficiencies. POEMS is an iterative optimization approach that repeatedly uses an evolutionary algorithm for finding the best modification to the current working solution, called prototype. Instead of directly evolving complete candidate concepts, POEMS focuses on using EAs for searching for the best set of modification of some initial working solution (or concept) so that the best discovered modification "*pushes*" the initial candidate towards the optimal concept.
The modifications are represented as a sequence that consists of elementary actions. The fact that each action can be coded into the form of numerical (integer) arrays enables simple algorithm implementation.
The EA used in POEMS searches for the best modification in this space of actions. It does not handle the solved problem as a whole, but is employed within the iterative optimization framework - its role is in each iteration to evolve the best modification of the current solution prototype. The load of searching for the best complete solution at once is cut into iterations, each of them representing a process of seeking the best partial transformation of the current solution prototype to the new and possibly better one.
One of the advantages of POEMS is that the standard selection, crossover and mutation operators can be used without any special modifications. Standard approaches that use EAs in combination with string data representation suffer strongly from problems caused by variation operators destroying links between elements when refining the discovered clauses. This leads them to development of highly specialized search operators.

The other advantage is that this approach enables for flexible and smooth incorporation of background knowledge into the search process and also enables for easy restricting the search space to legal hypotheses e.g. with utilization of context (used in this thesis).
Consideration of context during the clause construction phase enables to utilize the knowledge about relations between predicates in the data and construct the clauses with respect to these existing links. In addition, utilization of actions can easily lead to reaching beyond the scope of first order logic (e.g. by introducing numeric constraints).

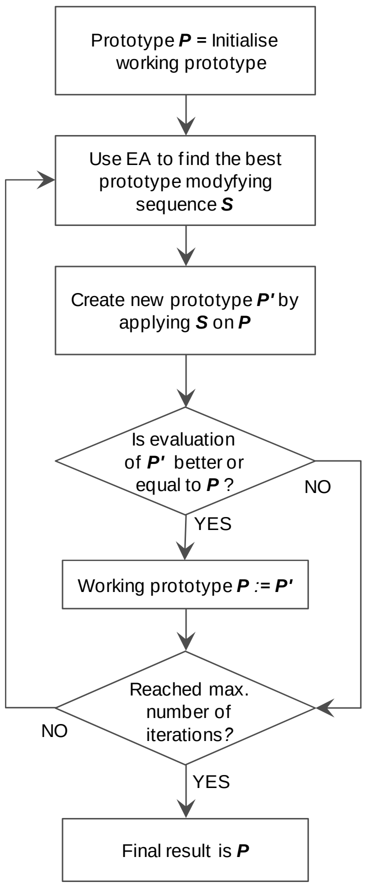

Prototype P = Initialise working prototype
Use EA to find the best prototype modyfying sequence S
Create new prototype P' by applying S on P
Is evaluation of P' better or equal to P ?
NO
YES
Working prototype P := P'
Reached max. number of iterations?
NO
YES
Final result is P

Figure 5.1 – Flowchart of basic POEMS algorithm

5.2 Basic POEMS Algorithm

The task of the EA optimization algorithm in POEMS is to find such sequence of modification actions that would adapt the working prototype in such way that the result would be optimal (or close to optimal) according to given criterion. Each modification is represented as a sequence of primitive actions and as such can be seen as a *meta-action*. The primitive actions are defined specifically for the problem at hand and come as an input parameter to the POEMS algorithm. When the best possible modification is discovered new prototype is created by applying the modification to the current working prototype. The search is repeated again from this new starting point. The problem for POEMS algorithm can be defined as follows:

Definition 5.1 (POEMS algorithm). Given working prototype P from the space of possible solutions S_P, a set S of primitive actions A, an action application function *apply(P, S)*: $S_P \rightarrow S_P$ that creates new prototype by applying the sequence of actions $S=[a_1,..,a_n]$, $(a_i \in A,\ i=1,..n)$ on P and a *fitness(Q)*: $S_P \rightarrow R$ function that specifies the quality of some prototype Q, the task is to find the sequence $S^*=[a_1,..,a_m]$ of elementary actions a_i that are selected from set of actions A such that

$$S^* = \arg\max_{S \subset A} fitness(apply(P,S)) \qquad (5.1)$$

This is equivalent to search for the optimal solution to given problem because the prototype $P'=apply(P,S^*)$ that results from application of S^* on the working prototype P is such solution.
It is usually impossible to directly find sequence S^* that would create the optimal concept from the starting concept as the starting concept is usually trivial and S^* can be very long. Therefore, in POEMS the search takes the form of an iterative process where in each iteration, EA searches for a shorter sequence of elementary actions/operations which improves the actual prototype concept the most (see Figure 5.1 for the algorithm flowchart).

Algorithm 5.1 The POEMS algorithm

```
function findBestPrototype
 Input: actions A, initial solution P
   Prototype = P
   Iteration = 0
   repeat
      Iteration++
      ActionSequence = run_EA(Prototype, A)
      NewPrototype = apply(ActionSequence, Prototype)
      if(fitness(NewPrototype) ≥ fitness(Prototype)
             Prototype = NewPrototype
   until(terminationConditionMet(Iteration, Prototype))
 return Prototype
```

After the EA finishes, the best evolved action sequence is checked for whether it worsens the current prototype or not. If an improvement is achieved or the modified prototype is at least as good as the current one, then the new (modified) prototype is considered as the working prototype for the next iteration. Otherwise, the current working prototype remains unchanged. The iterative process stops when it meets the stopping criterion which usually is reaching a specified number of iterations. An outline of the POEMS algorithm is shown in Algorithm 5.1 and Figure 5.1.
As can be seen from the algorithm description each iteration of POEMS algorithm can be divided into following three basic steps:

1) *Sequence optimization.* Running the optimizer (EA) to search for best prototype modification (each modification consists of a sequence of several parameterized instances of pre-defined atomic actions).
2) *Sequence application.* Application of the evolved action sequence onto the current working prototype solution.
3) *Evaluation of the new prototype.* After the evaluation of the new prototype a selection is made - better of the two prototypes (original and new) is used as prototype for the next iteration.

The three steps described above are repeated until termination condition is satisfied, (i.e. usually until specified number of algorithm runs is taken). In this manner the EA in POEMS is evolving sequences of adaptations to improve the working prototype concept and it does not evolve the complete concept as a whole.

Example 5.1. Consider a simple example depicted in Figure 5.2. The starting phenotype is labeled as a, final solution as b. If the solution is not reachable in single optimization iteration, the algorithm tries in each iteration to alter the start phenotype so that it follows the gradient of the given evaluation function and tries to "push" the prototype closer to the optimal solution. The iterative search process can thus be seen as a path on which the algorithm "moves" the prototype solution from the starting point closer to the optimum. If the algorithm is designed so that it does not search (or discards) the non-feasible solutions it is likely to follow the path with 3 intermediary solutions (path II). Path I cannot be realized as it leads through non-feasible intermediary solution x that cannot be used them as one of the prototype solutions.

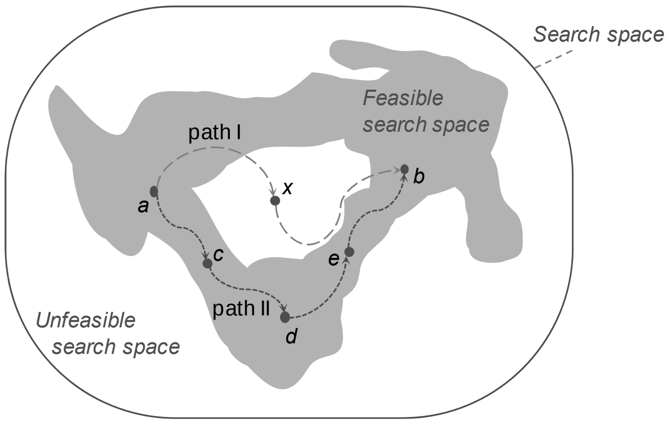


Figure 5.2 – Example of POEMS search path from starting prototype a to final solution b: path I (2 iterations with one temporary solution x) uses not feasible solution and cannot be produced by POEMS, path II uses only feasible solutions (4 iterations with 3 temporary solutions c, d and e)

An interesting aspect of the POEMS system was described in (Kubalik and Faigl 2006). Not only that it functions as a global explorer in early stages of the run but it converts to fine tuning of the prototype solution later on. This is due to the fact, that as the prototype gets better and better, it becomes very hard for EA to evolve an action sequence that would both improve and considerably change the prototype. Instead, trivial action sequences that do not modify the prototype much are produced. On one hand this points at decrease of the search abilities of the algorithm in higher iterations. On the contrary this shows that the searcher is capable to locally refine existing solutions that is favorable e.g. for solving ILP tasks.

5.3 Sequence Optimization in POEMS

The size of the space of possible modifications of the current prototype (the "*search radius*") explored by the EA at each iteration is determined by the set of available elementary actions and the maximum allowed length of evolved action sequences. The less explorative the actions are and the shorter the evolved sequences are the narrower neighborhood around the prototype is investigated and vice versa.

5.3.1 Chromosome Representation and Translation

The EA in POEMS evolves linear chromosomes of fixed length that are made up by a sequence of sub-strings – *codons*. Each codon represents an instance of certain action chosen from a set of given (user-defined) actions that modify the prototype and its parameters. The chromosome therefore represents a sequence of atomic actions that altogether form one *meta-action*. Thereby, the algorithm is evolving these *meta-actions* that act on the prototype, it does not evolve directly the problem solution. Each primitive action in the set of actions at hand is represented by a record, with an attribute *action_type* followed by parameters of the action.

Algorithm 5.2 - Chromosome translation in EA in POEMS

```
function translateChromosome
 Input: actions A, working prototype P, chromosome C
   NewPrototype = createCopy(P)
   repeat
      Codon = getNextChromosomeCodon(C)
      ActionID = getFirstField(Codon)
      Action = getActionInstance(A, ActionID)
      ActionParams = getActionParameters(Codon)
      NewPrototype = applyAction(NewPrototype, Action, ActionParams)
   until(endOfChromosome(C))
 return NewPrototype
```

Each chromosome represents one individual - one sequence of actions where every action is defined by one codon. Codon is a string that consists of action id and a set of parameters which immediately follow. The a*ction id* parameter defines the action instance that will be used. The other parameters are used to adapt the action outcome; the number of action parameters is usually constant for each specific problem. Yet, not all of them necessarily need to be defined or can be omitted during translation. Hence, the effective number of parameters for actions implemented in the problem side can easily vary (see Figure 5.3 for an example of chromosome). Thereby, one can use same codon length for actions that require different number of parameters. The pseudo code of the process of translation of chromosome to set of actions and creation of its respective prototype is given on Algorithm 5.2.

Example 5.2 *(Chromosome Translation in Subgraph mining).* An example of chromosome translation is depicted in Figure 5.3. The problem solved in this example is the search for interesting subgraph patterns in database of undirected graphs with labeled nodes and edges (e.g. a search for small sub-graphs that appear as parts of larger graphs and can be used to construct patterns). There are two actions that can be used in this example:

1) *addNode()* – this action adds one node to the evolved subgraph prototype (without connecting it to the subgraph); it needs one parameter that specifies the label of the node that is added by the action to the graph; the set of labels that may be used is *{a, b}*;
2) *addEdge()* – this action adds an edge to the working graph, three parameters need to be specified – id of the first node, id of the second node and label of the edge that is added; the set of labels that may be used is *{1, 2, 3}*.

The starting (working) prototype in this example is a graph that consists of three nodes that are connected by two edges to form a string *a*-*b*-*a*, both edges are labeled with '*1*' (see "*start phenotype*" in the figure – bottom left hand corner). The chromosome to be translated starts with numbers '[*1,2,5,1,2,3,4,3*]'.

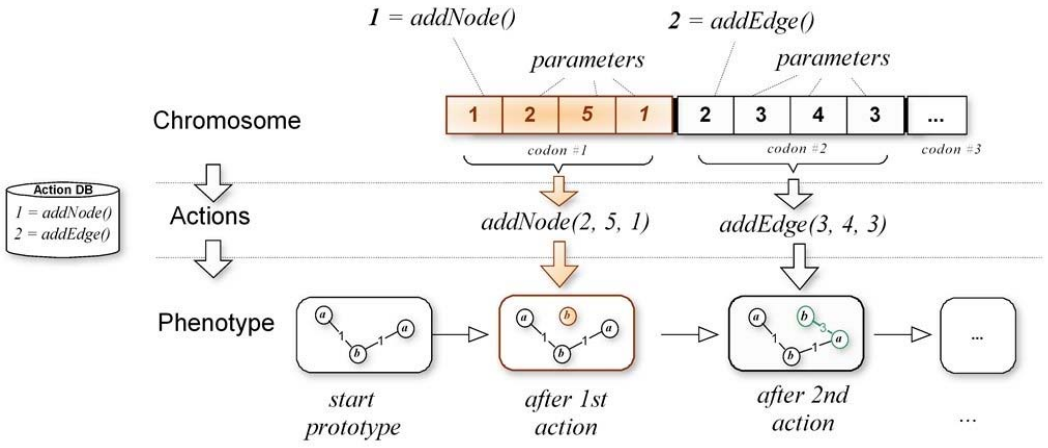


Figure 5.3 - Chromosome translation in POEMS algorithm

Because the maximum number of parameters needed by the specified actions is three, the chromosome consists of codons with length of four genes (where the first element of the codon identifies the action). The translation of the chromosome starts with the first codon, i.e. with [*1, 2, 5, 1*] (colored in Figure 5.3). The id of the action specified by the first codon is *1*, the parameters of the action are [*2, 5, 1*]. Hence the action *addNode(2,5,1)* is applied onto the current prototype. Using the first parameter

('*2*') results in adding a node that is labeled with second label from the set, i.e. with label *b*. Parameters '*5*' and '*1*' are not used by this action and therefore are omitted.
The translation then proceeds to the second codon, that bears numbers [*2, 3, 4, 3*]. The action id here is *2*, therefore action *addEdge* is used. The result of application of *addEdge(3,4,3)* is connection of third and fourth node in the graph with an edge labeled as '*3'*. After the two actions, the prototype has grown to the form '*a-b-a-b'*, the first two edges labeled as '*1*', the last one labeled as '*3*' – this is the node that was added by applying the actions coded by this chromosome.

5.4 Summary

In this chapter we have introduced the algorithm called ***Iterative Prototype Optimisation with Evolved Improvement Steps*** (POEMS). The POEMS system iteratively improves the prototype solution so that in each iteration an evolutionary algorithm is used to search for a sequence of actions, which would improve the current prototype solution. The main idea behind the algorithm is that starting with some initial prototype solution, the search proceeds by improving it in an iterative process. In each iteration, the most suitable modification of the current prototype is sought by standard optimization algorithm (evolutionary algorithm in our case).
This action-based approach is applied in such a way in which the EA evolves linear chromosomes of specified length, where each codon represents an instance of certain action chosen from the set of elementary actions defined for the given problem. A chromosome can contain one or more instances of the same action. Each action is represented by a record, with an attribute *action type* followed by parameters of the action.
In contrast to standard EAs, the genetic operators and the evolutionary model do not have to be modified to fit to the problem at hand. The framework allows using the basic variation operators such as standard 1-point, 2-point or uniform crossover and a simple gene modifying mutation. The problem specific task lies only in the action utilization. The user has to properly choose the set of actions and design the mechanism of application of the evolved action sequences. The actions should be chosen so that the space of possible candidate action sequences is rich enough to ensure sufficient exploration capabilities of the whole system.

6 POEMS for ILP

In this chapter we describe our approach to implementation of the POEMS algorithm into the ILP domain. After summarizing the motivations we present the basic three- layered structure of our system. Based on these layers we start the description with the model building layer where the classification model is constructed from the clauses generated by the POEMS algorithm. The next subsection explains how POEMS is used to construct the clauses for the model. Finally we describe the last layer that represents the refinement-action based interface between the EA and the ILP task described in FOL. We also give details on utilization of context and on the method of its automatic induction from given data.

6.1 Introduction and Motivation

Motivations for using the randomized search algorithms including Evolutionary algorithms (EAs) to solve ILP problems were already described in Chapter 4. The EAs represent a group of randomized techniques successfully used for solving difficult tasks. Yet, as the mentioned chapter also shows, the implementation of EAs in the field of ILP is not straightforward. Any effort in implementation of EA in the field of ILP must solve the basic problem of transforming the ILP task from FOL into a form where the EA can reasonably use its evolutionary search operators (e.g. convert the task to representation that optionally uses linear chromosomes or tree structures).
Each such solution shall be able to tackle the three main issues:

1) efficiently refining candidate solutions and efficiently searching the space of concepts - focusing search on candidate concepts that are neither too general (cover all given training examples) nor are meaningless in the problem domain (semantically invalid, do not cover any example);

2) efficiently evaluating quality of candidate solutions (calculating coverages in reasonable time) and tackling the plateau phenomenon;
3) exploiting the domain knowledge.

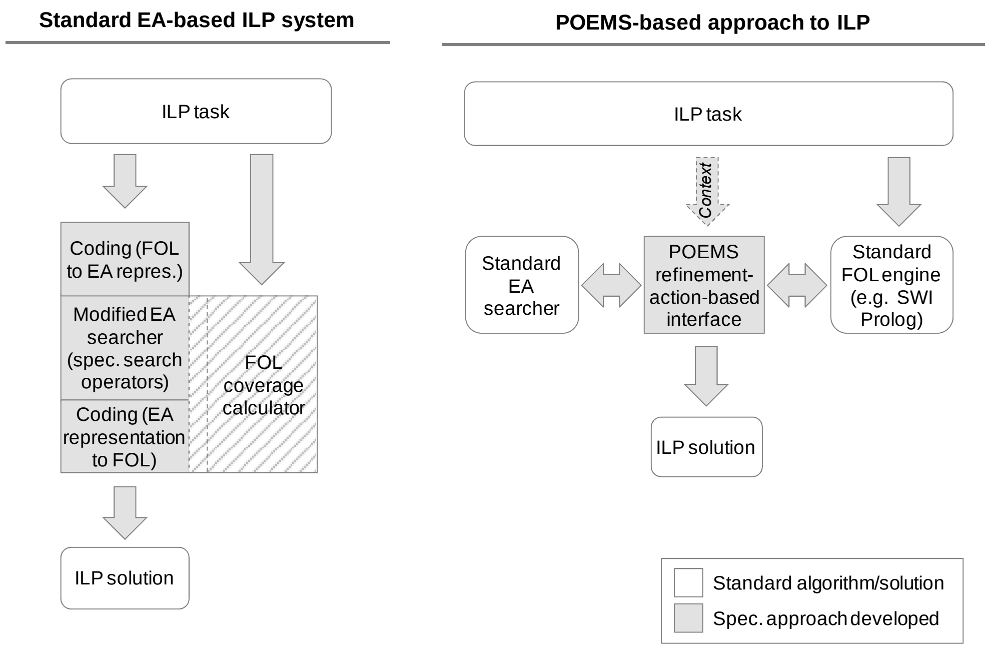

Standard EA-based ILP system
POEMS-based approach to ILP
ILP task
Coding (FOL to EA repres.)
Modified EA searcher (spec. search operators)
Coding (EA representation to FOL)
FOL coverage calculator
ILP solution
ILP task
Context
Standard EA searcher
POEMS refinement-action-based interface
Standard FOL engine (e.g. SWI Prolog)
ILP solution
Standard algorithm/solution
Spec. approach developed

Figure 6.1 – Comparison of standard EA-based ILP approaches to POEMS (cross-hatched block depicts standardized FOL module for which, however, often a specialized interface needs to be developed)

Apart from these issues we want our algorithm to fulfill two other targets that are often missed by the EA-based ILP algorithms which make these systems to be EA-hybrids highly specialized on ILP tasks. This stands in contrast to one of the basic characteristics of EAs – the domain independency. The two issues are:

1) utilization of a simple problem representation - all current EA-based ILP searchers use specialized representation of the problem to be able to solve the task by the EA system; these representations do not often fully allow using all capabilities of ILP refinements and additionally require development of specialized search operators;
2) full utilization of ILP capabilities – do not use some restricted version of ILP to solve the ILP problems (e.g. user defined clause templates, restriction on the type of refinement operators available etc.) but exploit all the possibilities of ILP.

According to us, an approach is needed that would enable utilization of all ILP refinement operators (e.g. including treatment of numeric values). In ideal case we would be able to use the EA “off the shelf” without any specific adjustments.
We believe that POEMS algorithm shall allow us achieving this target because it allows us to solve the ILP problem while keeping the form of the solution (clause) unrestricted and also offers utilization of all possible clause refinements available in ILP.

A simple scheme of the algorithm we realized in this thesis and its comparison to standard EA-based ILP systems is given in Figure 6.1. The figure shows that standard solutions usually adjust the whole EA to fit to the ILP problem. On the contrary, our approach based on refinement-action interface allows using both the EA and the FOL engine without any special adjustments.

6.2 POEMS for ILP Tasks

In the design of our EA-based system we use the POEMS algorithm as described in the previous chapter. In our adaptation of the algorithm for the purposes of solving ILP tasks the prototypes that are further adjusted by the action sequences take a form of logical clauses (described in FOL). The actions we use for this purpose are standard ILP clause refinement operators.
Like many other ILP systems, we do directly construct the whole classification theory (model) in one step. The model is constructed by a greedy set-covering approach and we use the POEMS algorithm to repeatedly search for single clauses for the model. Within this strategy, the classification model is built in several runs of the POEMS algorithm. The output from each run represents a partial solution to the problem that covers part of the training examples. This greedy approach to induce the classification theory is used by many inductive logic programming systems (the Pittsburgh approach).
To explain the function of our system we use a structure of three layers (as depicted in Figure 6.2):

- the first layer represents the action-based interface between ILP data and the searcher,
- on the second layer clauses are produced by the POEMS algorithm and
- on the third layer the ILP model is assembled from the clauses discovered on the second layer.

The basic flow of the algorithm for each of these layers is depicted in Figure 6.3.
In this subsection we will provide brief overview of the purpose of each layer. The details on the functionality of each layer follow in the next sections.

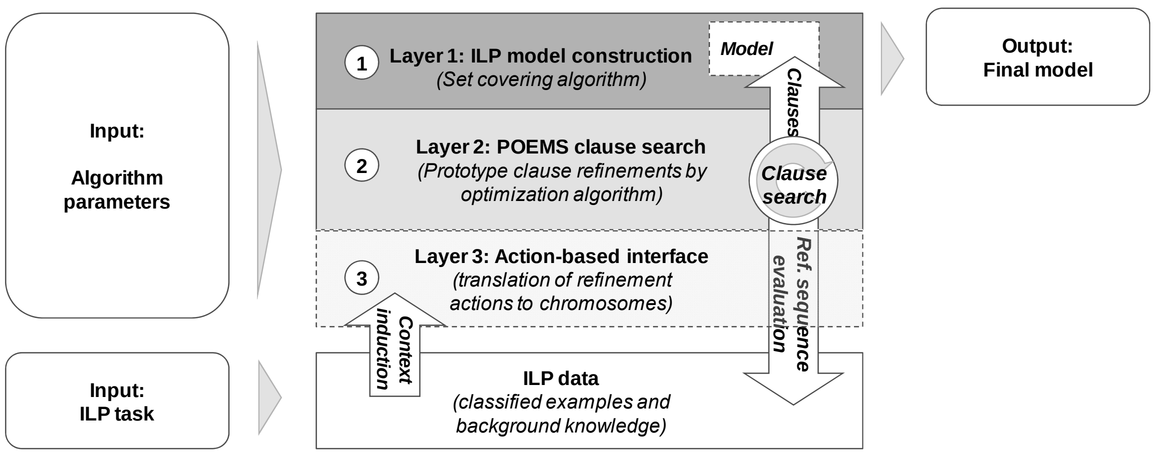

Input:
Algorithm parameters
Input:
ILP task
1 Layer 1: ILP model construction
(Set covering algorithm)
Model
2 Layer 2: POEMS clause search
(Prototype clause refinements by optimization algorithm)
Clause search
Clauses
3 Layer 3: Action-based interface
(translation of refinement actions to chromosomes)
Ref. sequence evaluation
Context induction
ILP data
(classified examples and background knowledge)
Output:
Final model

Figure 6.2 – Simplified structure of POEMS implementation for ILP tasks

Layer 1 – Model Construction. The purpose of this layer is to produce the final classification model. The model has a form of a set of n hypotheses where each hypothesis consists of a clause and example class associated with it:

$$Model = \{H_1, .. H_n \mid H_i = Class_j \leftarrow Clause_i,\ Class_j \in \{\text{Example } classes\}\},$$
$$H_i = Class_j \leftarrow C_i$$

where C_i is the clause and $Class_j$ is the class from which C_i covered most examples of the training set.

As we can see, the model represents a decision list - an ordered list of *if-then-else* rules that must be applied in sequence when classifying a new instance and as such is similar to approaches taken e.g. by (Riquelme et al., 2000).

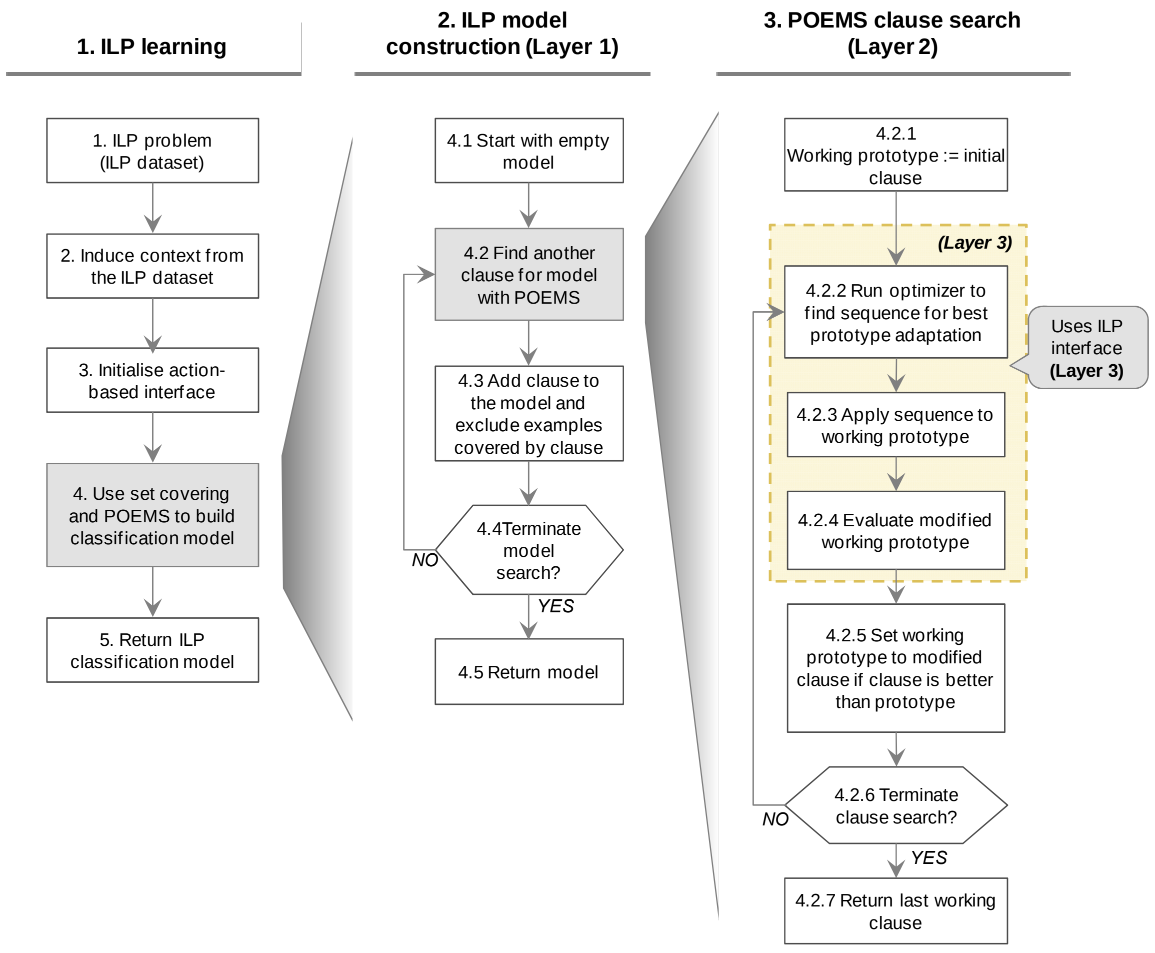

1. ILP learning
2. ILP model construction (Layer 1)
3. POEMS clause search (Layer 2)
1. ILP problem (ILP dataset)
2. Induce context from the ILP dataset
3. Initialise action-based interface
4. Use set covering and POEMS to build classification model
5. Return ILP classification model
4.1 Start with empty model
4.2 Find another clause for model with POEMS
4.3 Add clause to the model and exclude examples covered by clause
4.4Terminate model search?
NO
YES
4.5 Return model
4.2.1 Working prototype := initial clause
(Layer 3)
4.2.2 Run optimizer to find sequence for best prototype adaptation
Uses ILP interface (Layer 3)
4.2.3 Apply sequence to working prototype
4.2.4 Evaluate modified working prototype
4.2.5 Set working prototype to modified clause if clause is better than prototype
4.2.6 Terminate clause search?
NO
YES
4.2.7 Return last working clause

Figure 6.3 – Basic flow of POEMS implementation for ILP problem solving

When the model is used for classification of new, previously unseen examples, it takes the hypotheses in the order they were added to the model and tests whether the respective example is covered by the hypothesis clause. Once a clause covers the new example, the example is classified according to the

class associated with the clause. Otherwise, next rule in the row is tested. If none of the rules cover the instance then the default rule – assignment to majority class - is used for classification.
For the model construction we use the greedy set covering approach (see Algorithm 3.2) to assemble the model from clauses that are iteratively provided by POEMS algorithm running on the underlying *Layer 2*. Each time the POEMS produces a clause, the clause and the respective class associated with it added to the model and examples covered by this clause are excluded from the example set. If the example set still contains some examples, POEMS (*Layer 2*) is run again.

Layer 2 – Search for Clauses. The purpose of *Layer 2* is to realize the search for the clauses that later enter into the classification model. We use the POEMS algorithm to search for the clauses that are optimal with respect to given evaluation criterion (or, to be more precise, to search for the sequence of refinement actions that changes the given initial prototype clause into its best possible modification). The criterion in the case of ILP classification tasks measures the classification ability of clauses and prefers those that are able to distinguish better between the classes (e.g. entropy function).
The basic idea here is to develop the clause through repeated cycles (iterative) each time searching for best possible adaptation of the prototype clause thereby building the sequence incrementally. Therefore, the optimization algorithm used by POEMS is run in several iterations, each iteration "*pushing*" the prototype clause closer to the optimal solution. The representation used to code the sequences is the representation that is most suitable for the optimization algorithm used in POEMS. In our approach, linear numeric arrays (chromosomes) are used. These chromosomes are converted to the action sequences on the underlying *Layer3*.

Layer 3 - The Action-based Interface. *Layer 3* is used by the algorithm as the interface between the ILP data (defined in FOL) and the optimization algorithm that is used to search for best modification of given prototype clause. Purpose of this layer is to convert the numeric arrays produced by the optimizer on *Layer 2* into action sequences so that these can be applied on the working prototype clause to see and evaluate the result of the suggested refinement.
The sequences are within our algorithm represented as linear strings (chromosomes). Each chromosome consists of a series of substrings – codons - each codon representing one action and its parameters. On this layer the action sequence is created by parsing the chromosome codon by codon assigning each codon its respective action and action parameters. The parameters play an important role as the actions in our system are context-sensitive. This context sensitivity of the refinement actions is very important as it enables to limit the space of generated clauses and focuses the algorithm to produce clauses that have higher chances to cover some of given examples. We restrict the refinement actions outcome by applying context sensitivity because the search in ILP is difficult and the search space is huge. The actions are optimized stochastically and, when unrestricted, the actions produce clauses that have no sense (cover no examples).
The context information that is used for this purpose is induced automatically from the data (examples and background knowledge) prior to the start of the whole learning process. In the context induction phase the dataset is analyzed to obtain all possible and impossible co-occurrences of literals (e.g. `in_group(X,Y),circle(Y)`) and also co-occurrences of literal and constant (e.g. `triangle(Y, up)`). This information is stored in a structured record and used any time when context-sensitive action

is applied to restrict the space of possible action outcomes (e.g. by limiting unwanted co-occurrence of some literals).

6.3 Detailed algorithm description

In the next subsections we provide more detailed description including the algorithm pseudo-codes for the individual layers. To explain properly the function of our algorithm we will proceed in a top-down manner: we first focus on the model construction layer (*Layer 1*), then proceed to the POEMS search layer (*Layer 2*) and eventually finalize with the description of refinement actions and context induction (*Layer 3*). Detailed information about the implementation and concrete structure of the system developed for the purpose of this thesis is given in the Appendix - Section A.2.

Table 6.1 – Summary of the parameters of POEMS-based system for ILP

Parameter type	**Parameter**	**Description**
ILP interface (*Layer 1*)	Set of refinement actions *R*	The set of actions from which the POEMS algorithm constructs refinement sequences by which it alters the prototype clause. Any possible ILP refinement operator can be used within *R*, the algorithm also enables to work with user-defined problem-specific refinement actions.
	Context information *CTX*	The information about possible relations of predicates and constants derived from the dataset used by the context sensitive refinement actions to limit the search space
POEMS (*Layer 2*)	Maximal POEMS iterations I_{max}	The maximal number of iterations the POEMS algorithm is run before the altered clause is passed to the algorithm output.
	Initial starting clause C_S	The initial prototype clause on which the POEMS algorithm starts its work in the first iteration. This can be not only an empty clause but also some randomly generated or user-defined clause.
Sequence optimization (*Layer 3*)	Refinement sequence length *L*	Defines the length of sequence of refinement actions. The length can be either fixed to some given number of actions or an interval within which the sequence length is randomly generated.
	Clause evaluation function F_E	The function that is used as the optimization criterion when searching for the best clause modification. This is usually a function that describes the classification power of clauses, e.g. entropy function.
	Optimization algorithm *OA*	The algorithm that is used in POEMS iterations to find the best sequence of refinement actions to change given prototype clause (genetic algorithm, random walk etc.).

6.3.1 Basic structure of the system and the parameters

First we briefly introduce the parameters that are used by the system and will be used also for the description of the function of each layer. Except the training example set E the algorithm takes three categories of input parameters that can be grouped into three classes according to the layer they belong to (see Table 6.1 for the details about the parameters):

1) ILP interface parameters – parameters that determine behavior of the interface layer; these parameters are set of refinement actions and context information that is automatically derived from the ILP data.
2) POEMS parameters – parameters that influence run of the iterations of the POEMS algorithm; these parameters include number of iterations and the initial starting clause.
3) Sequence optimization parameters – the parameters that define type of the optimization algorithm, the evaluation function used in optimization and the length of evolved action sequence.

6.3.2 Layer 1: ILP model construction

In this section we describe the top layer (*Layer 1*) of the algorithm where the ILP model is assembled. In our implementation we use for this purpose the set-covering algorithm. The flow of model construction is depicted in Figure 6.4 and also described in Algorithm 6.1 below.
The ILP model construction layer takes – except the ILP data – two input parameters:

1) Maximal model size S_M – limit on the maximal number of clauses the model can contain.
2) Minimal clause coverage CVG_{min} – limits the minimal number of examples any clause of the model has to cover to be added to the model.

Given the set of classified training examples E (= the ILP task), we start with an empty classification model M and repeatedly use the POEMS algorithm to search for an interesting clause that would have the best classification characteristics on current set E (measured by user-defined clause evaluation function) and also that satisfies the minimal coverage criterion CVG_{min} (due to potential overfitting).
Once such clause C is discovered the next step is clause generalization. Motivated by Occam's razor principle with the aim to limit overfitting, clause C is replaced by its mos t simple generalization with the same or better value of the evaluation function (yet, this could theoretically lead also to preferring more complex clauses if the e.g. evaluation function prefers clauses with more literals). In this generalization step the algorithm greedily deletes constants and literals from the clause until the value of evaluation function for the clause improves or remains the same.
After the generalization step the algorithm identifies the largest class of examples from E that C covers ($Class_C$) and the hypothesis $Class_C \leftarrow C$ is added to the model ($Model = Model \cup \{Class_C \leftarrow C\}$). Finally, the examples covered by C are excluded from set E ($E = E \setminus E_C$, $E_C = \{ e \in E \mid B \wedge C \models e \}$).

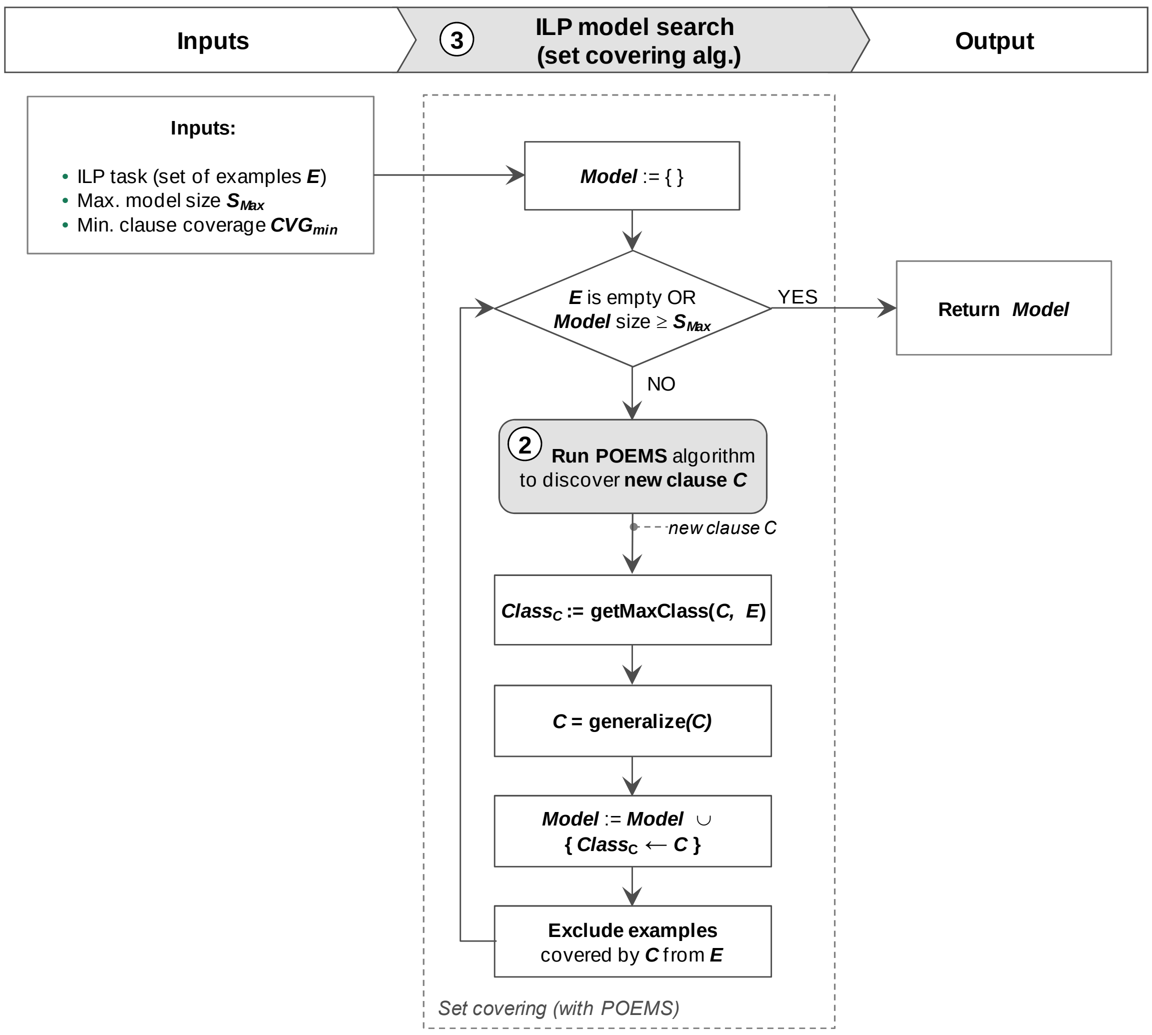


Figure 6.4 - Flowchart of *Layer 1* - set-covering algorithm used for ILP model induction in combination with POEMS algorithm

The whole clause construction loop is run until the number of clauses in the model reaches the given model size limit S_M (measured by the number of clauses the model contains) or until the example set E is empty. After the model search is finished last rule is added: an empty clause is added to encapsulate the model and to classify any possible example that would remain uncovered to the majority class.

Algorithm 6.1 – Building ILP theory with set covering algorithm using POEMS

```
Inputs:
- set of ILP classified examples E, maximal model size S_Max, minimal clause coverage CVG_min
- set of clause refinement actions R, context information CTX, maximal number of iterations I_max, starting clause (prototype) C_S, optimization algorithm OA, clause evaluation function F_E, length of refinement sequence L

function createModelWithPOEMS
Model = {}
  while(size(E)>0 and size(Model) < S_Max)
     BestClause = findBestClauseWithPOEMS(E,R,CTX,C_S,I_max,OA,F_E,L,CVG_min)
     If (coverage(BestClause) < CVG_min)
        Continue
     GeneralisedBestClause = generalize(BestClause)
     ClauseClass = getMajorityClass(GeneralisedBestClause, E)
     Model = Model ∪ [ClauseClass ← GeneralisedBestClause]
     E = E \ identifyExamplesCoveredByClause(E, BestClause)
  repeat
  Model = Model ∪ [MajorityClassificationCriterion ← EmptyClause]
return Model
```

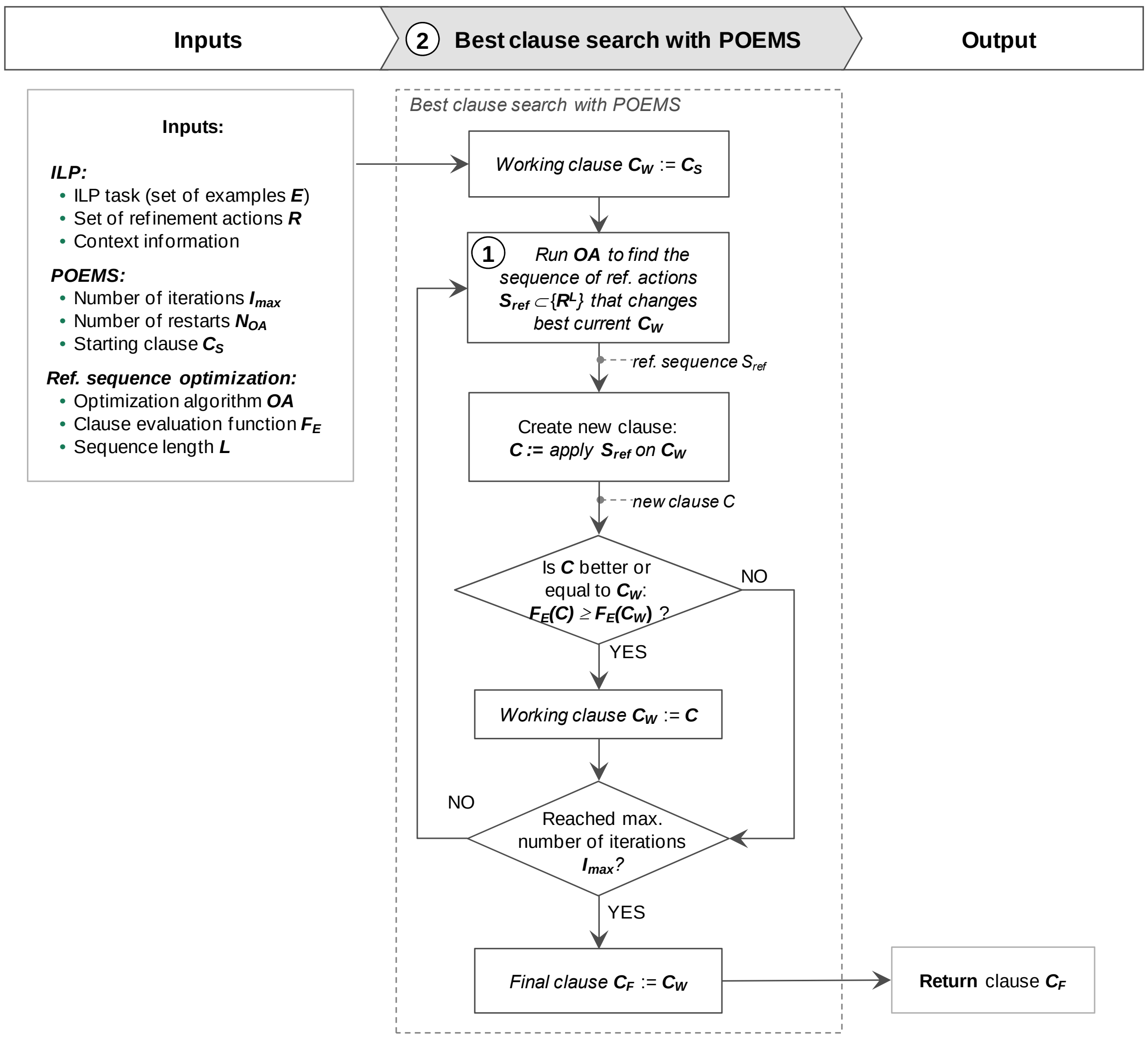
6.3.3 Layer 2: Search for the Clauses with POEMS

This section addresses the middle layer of our algorithm (*Layer 2*) where the clauses for the model are constructed by means of the POEMS algorithm. Our utilization of POEMS for ILP tasks follows basic principles of the algorithm as described in previous chapter (Chapter 5).

The flow of the POEMS algorithm we use for ILP is depicted in Figure 6.5. Starting with the given initial working prototype clause (e.g. with an empty clause), the working prototype is further improved in an iterative process. In each iteration, the most suitable sequence of clause refinements to the working prototype is sought by an evolutionary algorithm. The goal is to reach the most discriminative concept currently possible.

The sequences of actions (adaptations) produced by the evolutionary algorithm in each iteration can be viewed as meta-refinements in the context of ILP. After each iteration, the best evolved meta-refinement is applied and checked for whether it improves the current prototype clause or not. If an improvement is achieved or the modified prototype is at least as good as the current one, then the new modified prototype is considered as the working prototype for the next iteration. Otherwise, the current working prototype remains unchanged. This iterative process stops after a specified number of iterations the search finishes and the discovered clause is passed to the output.

It should be noted that replacing the prototypes in case of equal fitness helps to face the plateau phenomena, i.e. situations typical for ILP when the evaluation function is constant in large parts of search space (see the section below). This actually represents a situation similar to SAT problem solving for which the GSAT algorithm was developed.



Inputs
2 Best clause search with POEMS
Output
Inputs:
ILP:
• ILP task (set of examples E)
• Set of refinement actions R
• Context information
POEMS:
• Number of iterations I_max
• Number of restarts N_OA
• Starting clause C_S
Ref. sequence optimization:
• Optimization algorithm OA
• Clause evaluation function F_E
• Sequence length L
Best clause search with POEMS
Working clause C_W := C_S
1 Run OA to find the sequence of ref. actions S_ref ⊂{R^L} that changes best current C_W
ref. sequence S_ref
Create new clause: C := apply S_ref on C_W
new clause C
Is C better or equal to C_W: F_E(C) ≥ F_E(C_W) ?
NO
YES
Working clause C_W := C
NO
Reached max. number of iterations I_max?
YES
Final clause C_F := C_W
Return clause C_F

Figure 6.5 - Search for clause in ILP with POEMS algorithm on *Layer 2*

The success of GSAT (Selman et al. 1992) is attributed to its ability to move between successively lower plateaus making it become a common feature of the state-of-the-art SAT solving algorithms. In the work (Selman et al. 1992) the authors empirically showed that when a plateau (solution that is better than any of solutions in its close neighborhood) is reached then, instead of terminating the algorithm, continuing the search by making non-deteriorating, simple random ‘‘*sideways*’’ moves dramatically increases the success rate of the solver. Motivated by this, a simple and efficient greedy local search procedure was proposed called GSAT. When GSAT becomes trapped at some locally optimal solution it switches to search using random sequences of sideways search moves referred to as plateau steps. These are local search steps which do not lead to a change in evaluation function value but still change the solution.

This is actually the same technique as we use in the POEMS algorithm. When the iteration is complete the system decides whether the discovered solution should replace the current working prototype. In our strategy the new solution is adopted not only when it is better than the current one but also when it has the same value of evaluation function (see Algorithm 6.2 below).

Algorithm 6.2 - Search for ILP clause with POEMS

```
Inputs: set of clause refinement actions R, context information CTX, maximal
number of iterations $I_{max}$, starting clause (prototype) $C_{Start}$, optimization
algorithm OA, clause evaluation function $F_E$, length of refinement sequence L

Output: modified clause WorkingClause

function findClauseWithPOEMS
  WorkingClause := $C_{Start}$
  repeat
    CandidateClauses := { }
    for(iteration:=0, iteration < $I_{max}$, iteration++)
       NewClause := findBestModification(WorkingClause, OA, R, E, $F_E$)
       CandidateClauses := CandidateClauses $\cup$ NewClause
       BestClause := pickBestClause(CandidateClauses, $F_E$)
       if($F_E$(BestClause) $\geq$ $F_E$(WorkingClause))
          WorkingClause := BestClause
    end for
  return WorkingClause
```

6.3.3.1 *Clause Evaluation*

While the size of the search space represents the reason why we want to use EAs to solve ILP, the second complexity source lies in evaluation of quality of the constructed clauses/models. In our approach we treat the search for the theory in ILP as an optimization problem where the criterion is an evaluation function which prefers complete and consistent theories. All the evaluation functions used in ILP take as input the coverage of the evaluated clause (number of positive and negative examples covered by the clause). However, calculating clause coverage in ILP is a NP-complete problem and represents one of the key sources of ILP task complexity.

Previous research has demonstrated that strong gains in efficiency can be achieved by using unorthodox subsumption algorithms as opposed to the standard resolution procedures provided e.g. by the Prolog engine (Kuzelka and Zelezny, 2009). Recent statistical performance studies of search algorithms in difficult combinatorial problems have demonstrated the benefits of randomizing and restarting the search procedure (Železný et al., 2006). To avoid the lengthy subsumption tests performed by Prolog, we are using the latest randomized restarted subsumption testing algorithm *RESUMER2* in our work as the coverage calculator.

RESUMER2 (Kuzelka and Zelezny, 2009) was designed to support efficient mining in large relational structures by enabling fast pattern evaluation. The algorithm is complete in that it correctly decides both subsumption and non-subsumption in finite time. A basic restarted strategy is augmented by

allowing certain communication between odd and even restarts without losing the exponential runtime distribution decay guarantee resulting from mutual independence of restart pairs.
The empirical tests of *RESUMER2* against both the standard engines like Prolog and the latest state-of-the-art subsumption algorithm *Django* (Maloberti and Sebag 2004) show that *RESUMER2* is the best option. *Django* was shown to outperform by orders of magnitude the subsumption testing mechanism used in ILP and represents one of the best state-of-the-art coverage calculators (the logic behind *Django* is that it converts subsumption into a constraint satisfaction problem (CSP) and solves it by state-of-the-art heuristic techniques).
However, *RESUMER2* performs comparably with *Django* for relatively small examples (tens to hundreds of literals) and for further growing example sizes, *RESUMER2* becomes vastly superior.

6.3.3.2 Clause Coverage Buffer

Several works exist on the topic of increasing coverage calculation efficiency in the ILP systems by trying to avoid potential redundancy of clause evaluations and also focusing on efficient clause/query reordering (or even re-formulation with cuts and *once/1* functions) so that the coverage calculation is faster (e.g. Blockeel, Dehaspe et al. 2002, Struyf and Blockeel 2003, Fonseca et al. 2007).
There are two reasons why for our system we treat the redundancy of evaluations as the most critical issue in clause evaluation:

- the *RESUMER2* engine already deals with issues connected with efficient subsumption proving (optimized e.g. also by the cut transformations);
- the POEMS algorithm in its search for the best clause modification randomly samples mainly the neighborhood of the current working prototype which causes many duplicities among the generated clauses).

We use a simple technique to optimize the evaluation process by ordering the literals in each clause and buffering the clauses that were once evaluated. Each time a clause is passed to the evaluating engine we first test whether the coverage has not been already calculated. In order to avoid expensive subsumption tests to check for clause equality when passing through the buffer we adopt a simple solution to this task – we order the literals of each clause by an ordering $\prec_L$ that combines the literal name, arity (number of arguments), constants and variables used in the literal (see definition of $\prec_L$ below). After the ordering we re-label the variables (using names sorted in an ascending order) in the order in which the variables appear in the re-ordered clause. Thus we can approximate the clause equality by just simply comparing the clauses with the built-in text comparator and browse through the clause buffer more efficiently. Although this solution does not completely avoid duplicities, it is fast to calculate and able to limit strongly the number of redundant coverage calculations (see Appendix A.1 for empirical evaluation of this solution).

Definition 6.1 (Literal ordering). In the literal ordering $\prec_L$ literal l_1 is smaller than literal l_2 ($l_1 \prec_L l_2$) if

1) $l_1 \prec_\alpha l_2$, where $\prec_\alpha$ is standard alphabetical ordering, example: $a(X) \prec_L b(Y)$,
2) arity(l_1) < arity(l_2), example: $a(X) \prec_L a(Y,Z)$,

3) the set of arguments of l_1 with "*unified*" variable name (unified means all variables but the anonymous variable '_' are renamed to '*X*') is alphabetically smaller ($\prec_\alpha$) than the set of arguments of l_2 (example: *a(X,c,_)* $\prec_L$ *a(Y,Z,_)* because *[X,c,_]* $\prec_\alpha$ *[X,X,_]*).

Example 6.1 (Literal ordering). By application of the ordering $\prec_L$ the clause

r(Z,X), p(Z, _), p(X,c), p(X,_,_)

takes the following form:

p(X1, _), p(X2,c), p(X2,_,_), r(X1,X2).

Although there are many ways to order a clause - e.g. the clause can be represented in a form of a tree and the ordering then results from the tree traversal, this solution requires no special clause transformations and utilizes only the built-in (optimized) text comparator of the programming language we use for implementation.

6.3.3.3 Tackling the plateau phenomenon

Heuristic search in ILP is prone to *plateau phenomena* (Alphonse et al. 2004). This term is used to describe situations, when the evaluation function that is used to prioritize the samples of the search space during the search is constant in its parts and the search in these areas goes more or less blind or can get stuck.

An often accepted solution is a utilization of a form of *look-ahead* (Blockeel et al. 1998). Whenever a refinement of the described type is considered, it is immediately followed by a further refinement on this new part of the pattern. Generally, this *look-ahead* can be considered as two consecutive refinements. Each time a pattern is refined the learner considers all the regular node refinements plus the refinements of the *look-ahead* and chooses the best one from this extended set of refinements.

Another technique to tackle plateau effect is a solution that comes out of the GSAT algorithm (Selman et al., 1992) developed to solve CNF formulas. The basic principle is that the searcher is allowed – with some predefined probability - to take one or more random steps towards a node in search space of the same quality as the current one. Enabling such step has been proved to give better results than stopping the search on the plateau. The POEMS algorithm in its substance adopts the "standard" solutions to the plateau issue:

– *Meta-operators* – POEMS is based on evolving *meta-operators*, that actually are another form of look-ahead operators that when used should increase the probability of crossing the plateau;
– *GSAT plateau moves* - POEMS replaces the working phenotype in case at least an equally good solution has been discovered (i.e. the same approach like GSAT, see previous section);
– *Evolutionary search effect* - the selection operator of the GA that is used in the sequence optimization layer does not prefer only the best fit sequences thereby it actually explores various parts of the search space (a benefit of GAs).

Despite the fact that POEMS actually includes all these techniques, the results of first experiments with more complex datasets showed that sometimes another incentive is useful to support the search. Therefore, we suggest enriching the evaluation function by a complexity factor that – in contrast to other standard methods – further improves the quality of bigger patterns instead of the smaller ones. The straight motivation for solution is that there is simply more place for adaptations in longer clauses. Applying the complexity factor causes the searcher to prefer larger patterns *when selecting among patterns with the same coverage* and thus helps to overcome the plateau problem. This comes out of the expectation that when considering two clauses of the same coverage, the clause that is more complex has higher probability to be closer to the border of the plateau than the simple clause.

The main principle of implementing this approach is adding a small factor to the evaluation function that helps to promote more complex clauses (clauses with more literals and constants) in the search. The role of the factor should be such that it comes into play only when comparing clauses with the same coverage. Therefore the complexity of clauses is introduced to the fitness function as additional factor weighted by (small) parameter γ. In this way we actually introduce a search bias towards longer clauses. The only restriction here is the necessity for this factor not to overweight the main quality measure that is based on the pattern coverage. The fitness function then has following modified form:

$$F'(C) = F(C) + \gamma \ . \ complexity(C), \qquad (6.1)$$

where C is the clause and *complexity(C)* is the complexity of C measured as the number of literals plus number of constants used in clause C.

An example of such modification is e.g. following adaptation of F-metric function

$$F(C) = \frac{(\beta^2+1)\cdot precision(C)\cdot recall(C)}{\beta^2\cdot precision(C) + recall(C)} + \gamma \cdot complexity(C), \qquad (6.2)$$

The value of the γ parameter is automatically set in the following way. Although all the fitness functions are continuous functions, the numbers of examples covered are integers. Therefore, it is possible to find the minimal values the fitness function can take and also the minimal difference between these values (usually the difference between fitness for full coverage and coverage of all but one example). Then the complexity coefficient is set in following way

$$\gamma = \frac{min\,(F(c_i)-F(c_j))}{L_{max}+1}, \qquad (6.3)$$

where the symbols have following meaning:

- γ is the complexity factor,
- c_i and c_j are possible coverages of example sets (integers between 0 and size of E^+ resp. E^-): $c_{i,j} \in \{[a,b] \mid a, b \in \mathbb{N}, a = 0, .. |E^+|, b = 0, .. |E^-|\}$,
- F is the fitness function: F: $\mathbb{N} \times \mathbb{N} \rightarrow \mathbb{R}$,
- L_{max} is maximum length of clause (parameter to avoid increasing size of the clause over all limits)

Such a way to modify the evaluation function is also suitable to be used with POEMS. By using it we can directly influence the search process on the level where the sequence is optimized.
However, when adopting this approach, we are aware that by incorporating preference for more complex patterns we expose the searcher to the risk of over-fitting the model. To solve this issue, we add one more operation prior to the step when the pattern is added to the model. Before adding the pattern we replace it with its most simple generalization with the same coverage (see the "generalization step" in Figure 6.4). This is a simple operation and presents almost no over-head – performed by greedy cutting of the predicates and constants from the clause and de-unifying variables as long as the clause coverage is unchanged.

6.3.4 Layer 3: Interface to ILP – Context Sensitive Refinement Actions

As already described in Chapter 5 the POEMS algorithm does not directly solve the problem by performing search in the space of potential solutions. Instead, it solves the problem by looking for best adaptation of some initial solution and searches the space of possible adaptation sequences. The classification concept modifications are in the ILP generally performed by means of the refinement operators (Muggleton 1991). These operators alter the hypothesis (or clause) by producing its syntactical modifications. (see Section 3.4.1.1 for definition of refinement operators).

Example 6.2 The '*Add literal*' action. To give an example of action utilization we show here a utilization of the "*add literal*" action in our Bongard example problem (see Figure 6.6 or Section 3.3.2 in chapter on ILP). Let us have the following initial clause *C* = '*group(X), in_group(X,Y)*', set of literals *L={circle/1, triangle/2, in_group/2}* and the refinement action to be performed is the '*addLiteral(3)*' action where '*3'* is the action parameter.
Because the action adds a literal to the initial clause, we take the set *L* and based on the action parameter we choose which literal will be added. Because the third literal in the set is '*in_group/2*', the final clause has the form *C'* = '*group(X), in_group(X,Y), in_group(W,Z)*'.

In our approach, any ILP refinement operator can be used within the set of actions from which the POEMS algorithm selects actions for the refinement sequence. We have implemented both the generalization and specialization operators (see section 3.4.1). While the specialization operators specialize the clauses and make them more complex (by adding literals, unifying variables or replacing variables by constants), the generalization operators work in the other direction and simplify the clauses (by deleting literals, anti-unifying variables or replacing constants with variables).
The set of refinement operators/actions that were used with the ILP problems in this work are given in Table 6.2 (context-sensitive actions) and in Table 6.3 (general context non-sensitive actions). While the group of context non-sensitive operators can be applied without any restrictions regarding the actual state of the clause they are applied to, the output of the context-sensitive actions is influenced by actual form of the clause and the position in the clause where the operator is applied to. If for any reason the selected action cannot be applied (e.g. predicate removal from a clause with no predicates or instantiation of variable in position that offers no constants) the action has simply no effect and the prototype clause remains unchanged (action behaves like the *nop()* action).

Table 6.2 – Context-sensitive refinement actions used with POEMS

Action	Description	Example of application

addLiteral(L,A,N)	Select literal L from the actual prototype clause and in the literal select its argument A. Add new literal N to the clause so that this literal is immediately bound to argument A of the prototype literal L	`in_group(X,_)`
		 change to
		 `in_group(X,Y),`
		 `circle(Y,W)`

instantiateVariables(L,A,C)	Instantiate variable in literal L argument A by constant C – replaces the variable currently placed in A by a constant C	`circle(Z,_)`
		 change to
		 `circle(Z,red)`

unifyVariables(V1, V2)	Select two different variables $V1$ and $V2$ from the prototype that can be unified and unify them	`in_group(X,_),`
		 `circle(_,_)`
		 change to
		 `in_group(X,Y),`
		 `circle(Y,_)`

addNonEq(V1,V2)	Select 2 of the variables used in the clause $V1$ and $V2$, add non-equality constraint $V1 \neq V2$	`in_group(X,Y)`
		 change to
		 `in_group(X,Y), X ≠ Y`

*addLTE(L,A,C)**	Add inequality constraint to variable in predicate argument – select variable V bound to argument A of literal L from the prototype and add the inequality $V \leq C$ where C is a numeric constant from given set (e.g. one of the numbers that appears in the given dataset on the selected position in the literal)	`size(X,Y)`
		 change to
		 `size(X,Y), Y ≤ 5`

*addGTE(L,A,C)**	Add inequality constraint to variable in predicate argument – selects variable V in argument A of literal L from the prototype and adds the inequality $V \geq C$ where C is a numeric constant from given set.	`size(X,Y)`
		 change to
		 `size(X,Y), Y ≥ 5`

* *The last two actions marked with asterisk were used only in problems that contained numeric data (integers, real numbers).*

Table 6.3 – General (context non-sensitive) actions used with POEMS

Action	Description	Example of application

addLiteral(L)	Select literal L from the available literals and add it to the clause without binding it with a variable	`in_group(X,_)`
		change to
		`in_group(X,_),`
		`circle(_,_)`

removeLiteral(L)	Select literal L from the actual prototype clause and remove it	`in_group(X,Y),`
		`circle(Y,_)`
		change to
		`in_group(X,_)`

deleteConstant(C)	Turn the specific constant C in an argument of the clause into variable – i.e. replace the constant with universal variable symbol '_'	`circle(Z,red)`
		change to
		`circle(Z,_)`

nop()	Take no action – skip the remaining part of current codon and proceeds with translation to next codon (~action) in chromosome	`circle(Z,_)` change to `circle(Z,_)`

6.3.4.1 *Translation of Chromosomes to Actions*

In POEMS, the sequences that the GA evolves are coded into linear arrays (chromosomes) where each sequential part of chromosome (codon) represents an instance of certain refinement action chosen from a set of given (user-defined) actions that modify the prototype and parameters of the respective action. Therefore, the length of the sequence is determined by the length of chromosome that is used to code it. Number of atomic actions in the sequence is equal to the number of codons of the chromosome. The size of the space of possible modifications of the current prototype explored by the GA in each iteration is determined by the set of available elementary actions and the maximum allowed length of evolved action sequences L. See Algorithm 5.2 for the pseudo code of the process of translation of chromosome to its respective clause.

Codons that code each refinement action consist of a record with an attribute defining the action type followed by parameters of the action. The parameters specify function of the action. For example, if the action type points to '*add literal*' action, the corresponding parameters that follow in the codon after the action type information define (by order they appear in the codon) which literal in the working clause is to be extended, which argument of this literal will be used for the extension and which literal is to be connected to this argument. In cases when the selected action cannot be applied to the actual working prototype, the action is skipped and has no effect on the prototype.

The actions we used in this work were described in the previous section (see Table 6.2 and Table 6.3).

Implementation of chromosomes. In our approach, the information about the refinement actions that are used to optimize the initial clause prototype is carried by the chromosomes. Each chromosome is a byte array - it consists of a sequence of codons each also taking form of byte array. Each codon represents one action (see Figure 5.3 in POEMS description). Because the refinement actions used in our work take up to three parameters, we used fixed codon size of four bytes (first byte to determine the refinement action, next three bytes to define action parameters). This means a sequence of 5 refinement actions is coded by a chromosome array of 20 bytes.

6.3.4.2 Context-dependent Refinement Actions

In this subsection we will describe our concept of context and its utilization with the context-sensitive actions. We use the context-sensitive actions to face the problem of enormous size of the ILP search space.

All standard evolutionary optimization algorithms (except GPs) work in their basic setting with linear data representation. The key problems when using the EAs in ILP are: generation of meaningful clauses at random, translation of the clauses into the form of arrays that the algorithms are designed to work with and subsequent meaningful modification of the clauses. The search space in ILP is large and the simple random clause generation cannot be used for clause search as almost all randomly generated clauses cover either no example or all of given examples. Efforts have been made to improve efficiency by restricting the form of a clause e.g. by some template but these solutions often require many constraints and are too problem-specific.

Due to the enormous size of the hypotheses space it is extremely inefficient to rely on the fact that necessary relations shall sometime “just appear” during the search. It is important to ensure that each literal of the constructed clause is properly related to the rest of the clause and that such relations are not immediately destroyed by the search mechanism. Therefore, when a refinement operator is applied to the prototype clause we use following two sources of information to restrict the set of action possible outcomes:

1) *position of the refinement* – identifies the place in the current clause where the refinement is applied;
2) *structure of the analyzed dataset* - the information about the relations that appear in the whole dataset (is induced prior to the learning process).

Position of the refinement. In our approach, application of each context-sensitive operator (such as adding a literal or replacing a variable with a constant) is always bound to specific variable and its location within the whole working prototype clause. In this way, the outcome of each refinement operator gets context dependent and we limit co-occurrences of unwanted literals or utilization of constants in undesired arguments.

To give an example - when a predicate is added to the clause, the set of all predicates that can be theoretically added is pre-filtered by selecting only those literals that are suitable for the selected position in the current clause. This forms a limited subset of predicates from which the final choice is made. In the situations when due to the context constraints the set of results for the selected is empty and the action cannot be applied (e.g. no literal can be added to given clause), the action has simply no effect and the prototype clause remains unchanged.

Dataset structure. The dataset structure is parsed in the initialization phase of the algorithm. In this phase the structure of the dataset (existing combinations of literals and constants) is automatically induced and stored in the form of a graph of relations (see definition of the graph below).

Definition 6.2 - Graph of relations G_R. We define the graph of relations G_R as $G_R = <V, E>$, where V is the set of nodes (each node corresponds to one argument of a literal or to one constant from the dataset) and E is set of edges. Two nodes n_1, n_2 are connected by an edge from E if

– n_1 corresponds to the i-th argument of literal L, n_2 is some constant that can be used in the i-th argument of L (example: in *'triangle(T,up)'* constant *'up'* can be used in the second argument of literal *triangle/2*) or

– n_1 corresponds to the i-th argument of literal L_1, n_2 corresponds to j-th argument of literal L_2 and the same variable can be used concurrently in both arguments n_1 and n_2 (example: in *'square(S) and inside(S,X)'* the variable *'S'* can be used concurrently in the first argument of *square/1* and first argument of *inside/1*).

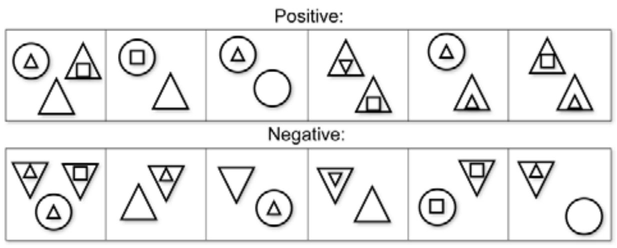


Figure 6.6 – ILP example – Bongard problem, sets of positive and negative examples

Example 6.3 – Graph of relations. To give an example of the *Graph of relations* we here present this graph for our Bongard example problem (see Figure 6.6 or Section 3.3.2 in chapter on ILP). The graph of relations of this dataset is depicted in Figure 6.7. Due to better transparency of the figure we grouped nodes that correspond to arguments of the same predicate together so that each predicate can be immediately identified.

A brief analysis of this graph shows that the constants '*up*' and '*down*' can be used only in the second argument of the *triangle/2* predicate. Because they are not used in any other argument, this argument is not connected (i.e. cannot share variable) with any other predicate from the dataset.

Also, according to definition above, there is no edge e.g. between the first (and only) argument of literal *circle/1* and first argument of *inside/1* because there is no circle inside of any object in our example. More graphs of relations of other problems can be seen in Figure 7.3 and Figure 7.4.

Although the context that is used for restricting the refinement actions can include more complex combination of predicates and constants, we consider only context with scope that includes relation of predicates-to-predicate or predicate-to-constant when creating the graph. Except the fact that this is the basic sufficient case for relational domains there are two computational reasons why to use this form of context:

– the number of queries that need to be performed over the data during context information induction rises exponentially with broadening the scope and
– utilization of larger restriction sets limiting possible refinements during action application gets extremely complicated when more complex form of context information is used.

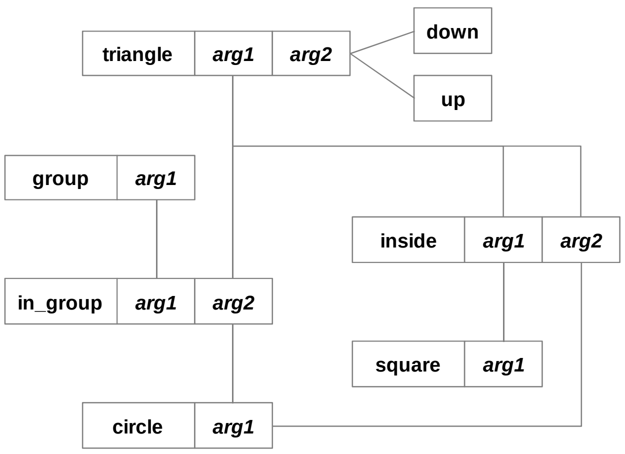


Figure 6.7 - Graph of relations of Bongard dataset from Section 3.3.2, nodes corresponding to arguments (noted as arg1 and arg2), lines signalize existing relation between arguments (nodes of one predicate grouped together for higher transparency)

Additionally, utilization of the context information obtained from the analyzed dataset in this form is sufficient to smoothly incorporate the background knowledge (because the graph of relations can also contain other predicates than only those utilized by the set of examples).

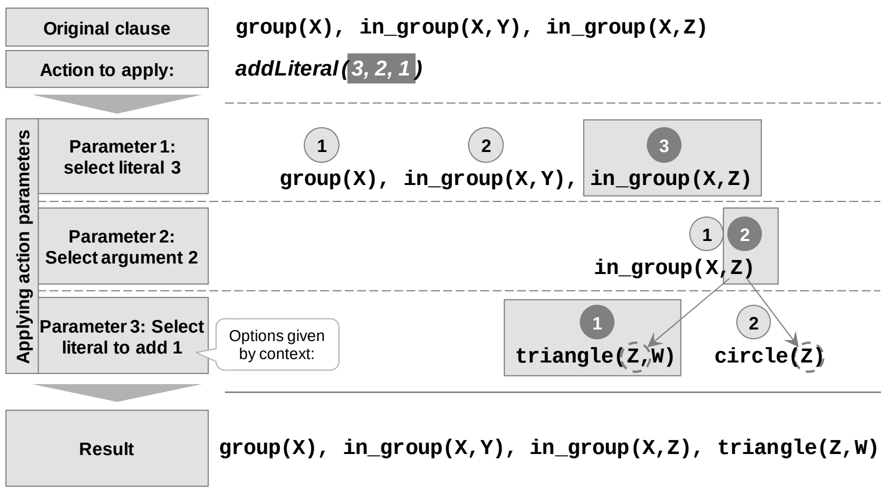

Original clause
group(X), in_group(X,Y), in_group(X,Z)
Action to apply:
addLiteral(3, 2, 1)
Applying action parameters
Parameter 1: select literal 3
1
2
3
group(X), in_group(X,Y), in_group(X,Z)
Parameter 2: Select argument 2
1
2
in_group(X,Z)
Parameter 3: Select literal to add 1
Options given by context:
1
triangle(Z,W)
2
circle(Z)
Result
group(X), in_group(X,Y), in_group(X,Z), triangle(Z,W)

Figure 6.8 – Example of application context sensitive action (applying '*addLiteral(3,2,1)*' to given clause *group(X), in_group(X,Y), in_group(X,Z)*

Example 6.4 Context-sensitive '*Add literal*' action. To give an example about how the context sensitive actions work we present here the context sensitive version of the '*add literal*' refinement action used with our Bongard example problem. Example of the action application is schematically depicted also in Figure 6.8.

Let us have the following initial clause '*group(X), in_group(X,Y), in_group(X,Z)*' and the action to be performed is the '*addLiteral(3,2,1)*' where '*3*', '*2*' and '*1*' are its parameters. Starting with the first parameter (the integer '*3*') we select the third literal to be extended (third literal in the table). This is '*in_group(X,Z)*'. Because the second parameter of the action is '*2*' the new literal will be bound to the second argument (number '*2*') i.e. to variable '*Z*'.
According to *Graph of relations* of this dataset (see Figure 6.7), there are only 2 literals that can be added to second argument of *in_group/2*. These are *triangle/2* and *circle/1*. The third parameter of the action is '*1*' and therefore the first literal from this set (number '*1*') will be chosen. This is the *triangle/2* literal. At the end, the final clause has the form '*group(X), in_group(X,Y), in_group(X,Z), triangle(Z,W)*'.
Note, that in our work we are using the *modulo* in situations when parameter value is higher than current number of possible choices. If that happens, using modulo after dividing the actual parameter value with known maximum value converts it back to the desired range.

Algorithm 6.3 - Context induction from ILP dataset

```
Input: ILP dataset $D$
Output: graph of relations in the data $G_R$

function induceGraphOfRelations
$G_R$ := empty graph of relations
$P$ := getSetOfAllPredicatesFromData($Data$)
$C$ := getSetOfAllConstantsFromData($Data$)
addNodesToGraph($G_R$, $P$, $C$)
for(each predicate $p$ from $P$)
   for (each argument $arg_{pi}$ of $p$)
      for (each constant $c$ from $C$)
         if(canUseConstantInPredicate($D$,$p$,$arg_{pi}$,$c$)
            addEdgeToGraph($G_R$,[$p$, $arg_{pi}$],$c$)

      for (each predicate $q$ from $P$)
         for (each argument $arg_{qj}$ of $q$)
            if(canShareTheSameVariable($D$,p,$arg_{pj}$,q, $arg_{qj}$)
               addEdgeToGraph($G_R$,[$p$, $arg_p$],[$q$, $arg_q$])
return $G_R$
```

Inducing the Graph of relations. To conclude this section we need to describe the context induction algorithm. The pseudo-code of the algorithm is given in Algorithm 6.3. The algorithm first creates an empty graph G_R and constructs sets of all constants C and set of all predicates P that are used in the supplied ILP data. Then it adds all constants and tuples [predicate, argument] as nodes into G_R. To add edges the algorithm takes each predicate p from P and tests each of its arguments arg_{pi} against ILP data for two basic relations (corresponding to Definition 6.2):

1) *relation with constants* – if a constant $c \in C$ can be used in predicate $p \in P$ in the place of its argument arg_{pi} an edge is added to the graph to connect the nodes [p, arg_{pi}] and c;

2) *relation with other arguments of other predicates* – if two predicates $p \in P$ and $q \in P$ can share the same variable in their arguments arg_{pi} and arg_{qj} an edge is added to the graph to connect the respective nodes $[p, arg_{pi}]$ and $[q, arg_{qj}]$.

6.4 POEMS for ILP: Summary

In this chapter we have introduced our approach to utilization of Evolutionary Algorithms in ILP. Motivated by the search complexity of ILP as well as by the issues of 'direct' implementation of evolutionary algorithms in ILP we decided to use the POEMS algorithm. The POEMS algorithm solves the problem by searching for the best sequence of modifications of some initial solution. Therefore, it does not directly solve the problem by performing search in the space of potential solutions but it searches the space of possible sequences of adaptations (refinements in the case of ILP). Using this approach allows us to solve the ILP problem while keeping the form of the solution (clause) unrestricted and using all possible clause refinements that ILP offers.

To describe our approach we have used the schema of three basic layers and for each layer we have described its function and the parameters. The classification model is constructed on *Layer 1.* We do not use POEMS algorithm to directly produce the ILP classification theories. Instead, we use the set-covering strategy, build the model clause by clause and utilize the POEMS algorithm to search for the individual clauses.

Layer 2 is the layer on which we run the POEMS algorithm. We start with given (usually empty) clause and use POEMS to iteratively search for such sequence of refinement actions that would modify the starting clause to a clause that has better classification properties (measured by given evaluation function). POEMS is run iteratively which means that the final sequence of actions is constructed by parts, each iteration slightly modifying the working clause. The sequences are represented in the form of numeric arrays (chromosomes) that are translated into refinement actions on the underlying layer.

Layer 3 is used to translate the chromosomes to action sequences. Because random search in ILP is complicated by the enormous size of the search space we use context-sensitive refinements. The context-sensitivity means that when a refinement is applied we restrict the set of possible outcomes by considering position of the refinement in the actual clause and also the structure of the whole dataset. This ensures that impossible literal-literal or literal-constant co-occurrencies will not appear in the clause, each literal of the constructed clause is properly related to the rest of the clause and that such relations are not immediately destroyed by the stochastic search mechanism. Except for describing the function of Layer 3, we also have introduced the possibility of extension of our algorithm by library-based refinements generated by means of automated graph mining algorithm.

It should be noted that the approach most similar to our work is the *Foxcs*-2 algorithm which is a learning classifier system approach to relational reinforcement learning (presented in Section 4.4.9). Exploration of the rule space in *Foxcs*-2 is principally achieved by applying ILP generalization and specialization refinement operators. However, in contrast to our approach, the system uses just one operator at a time. It also does not evolve sequences of these operators nor does it restrict their function according to the state of adapted clause.

7 Experimental Evaluation

In this chapter we present the experimental evaluation of the various components of our POEMS-based ILP system that were introduced in the previous chapter. Following the basic structure of three basic layers of the algorithm (see Figure 6.2) we want to experimentally confirm our hypotheses about the effects of the most important parameters on the algorithm performance and to find optimal setup of the system. For each of the parameters we defined the hypothesis that we want to test, the experiment and criteria by means of which we are able to do so. We proceed in a bottom-up manner: starting with *Layer 3* (context), proceeding to *Layer 2* (sequence optimization parameters) and eventually reach the top level *Layer 1* (model construction parameters).

We will analyze both the standard parameters of the POEMS implementation in ILP (e.g. maximal model size, number of POEMS iterations etc.) as well as the impact our suggestions to make the learning process more efficient (e.g. context sensitivity or clause generalization). For the standard parameters of the POEMS system that are used in our system we want to learn their optimal setup and investigate, how the setup affects both the quality of the solution found (accuracy of the final model) and the search complexity needed for the computations. For the suggestions introduced in this work that should help to reduce the runtimes we need to evaluate whether and how much they influence the quality of the final solutions and the computational time required by the learning process.

In our experiments we use five ILP datasets, four of which are considered to be standard ILP problems. We use these data not only for validation of our concept and for setting the parameters but also for benchmarking with other state-of-the-art ILP algorithms.

The chapter is structured in the following way: first we describe the settings of the experiments, the datasets and the methodology used. Then we proceed in a bottom-up manner from *Layer 3* to *Layer 1* with the aim to evaluate the effect of respective parameters, analyze the performance of our system and the computational effort needed under varying circumstances. The last section of this chapter focuses on comparing the results of our system with results published in the literature.

7.1 Set-up of the Experiments

In this section we describe the set-up of the experiments, provide brief information about the datasets used, the performance evaluation methodology and the refinement actions used in the experiments.

7.1.1 The Datasets

We carried out the experiments with the following five ILP datasets:

- *Trains* dataset (Michalski 1980),
- *PTE* Mutagenesis (Srinivasan et al. 1999),
- *PTC* Toxicity (Helma et al. 2001),
- *CAD* dataset (Žáková et al., 2007) and
- *Alzheimer* dataset (King et al., 1995).

These datasets were selected mostly because they present standard datasets that are used for ILP benchmarking, except from the *CAD* dataset that represents a real-world engineering problem. The choice was motivated by the effort not to perform experiments only on simple artificial dataset (*Trains*) but also test the performance on real data and examine noisy and larger datasets. Additionally, the last dataset (*Alzheimer*) represents a relation learning issue selected to show that our system is able to handle different type of tasks then only pure classification as well.

Trains dataset. The simplest dataset used in this work is the well-known "*Trains Going East or Going West*" problem proposed in (Michalski 1980). This dataset represents standard artificial problem of learning in FOL and was primarily constructed to illustrate ILP learning capabilities. Although it has only few instances it allows for testing the basic properties and convergence of relational algorithms. The basic version consists of 10 examples, each example describes one train, five trains are classified as westbound, five as eastbound. For the description of each train, predicates from a set of 10 different predicates are used (these are *infront/2*, *has_car/2*, *long/1*, *short/1*, *closed/1*, *open/1*, *shape/2*, *wheels/2*, *load/3, train/1*). The original *Trains* problem of Michalski has a very simple solution with 3 literals (*eastbound(X1) ← has_car(X1, X2), closed(X2), short(X2)*). Therefore, for some experiments we have used stochastic train generator to create more complex problem. Our two sets of 10 trains were constructed so that the shortest clause separating them into 2 groups (10 trains each) has the length of 7 literals (tested by exhaustive search in Aleph). As the dataset is small we do not use the cross-validation with this dataset (as we do with other datasets) and analyze it only as a whole.

PTE Mutagenesis. The second dataset is the well-known *PTE mutagenicity problem* dataset (Srinivasan et al. 1999). The problem belongs to the group of Structure-Activity Relationship (SAR) tasks in which relational learning is used to explain the characteristics of given molecules based on the chemical patterns these compounds contain. The aim is to discover clauses that distinguish mutagenic chemical compounds from non-mutagenic compounds in the given set of 188 different chemical molecules that are classified into two classes (125 mutagenic and 63 non-mutagenic). Compared to the *Trains* problem presented above, in the *PTE* problem the data is far more complicated (each example contains up to hundred of literals) and the assignment to classes is less straightforward (no classifier that would work with 100% precision has been found yet).

Table 7.1 – The *PTE* dataset, basic parameters

Number of predicates	atom/3	4 706 occurrences
	bond/4	5 243 occurrences
	other structural predicates	25 different predicates, 1 107 occurrences in total
	numeric predicates	2 predicates for each of 188 compounds
Number of examples in classes	Positive compounds	125 examples
	Negative compounds	63 examples

Two basic relations are used to represent molecule structure: *atom/5* and *bond/4*. Each molecule is also described by a flag *ind/2* signalizing presence of three or more fused rings in the compound. Additionally, information about presence of specific atoms in one of 23 defined chemical substructures (like ketone- group, amine- group etc.) is available. Except from the standard labeling of bonds and atoms that uses categorical labels, compounds are described by two real-valued attributes - the energy of the lowest unoccupied molecular orbital (*lumo/2*) and the molecular hydrophobicity (octanol/water partition coefficient *logP/2*). This is also the reason why we chose this dataset as we want to show that our system is capable of handling the numerical data as well. For more information about the dataset see e.g. (Srinivasan et al. 1999).

PTC Toxicity. The task of our third dataset - the Toxicology prediction of chemical compounds (*PTC*) - is to classify chemical compounds by carcinogenicity and as such represents another SAR task. The *PTC* data set consists of 417 compounds with 4 types of test animals: male mouse (MM), female mouse (FM), male rat (MR) and female rat (FR). The data contain following elements:

– the structural features of the molecules:
  a) basic compound structure - atoms and bonds;
  b) information about specific atom groups (nitro-, alkene- etc.) and rings (aromatic, non-aromatic);
– the carcinogenicity classifications - the National Toxicology Program carcinogenicity classification labeling was used to classify these compounds, each molecule is classified into one of eight classes ("*ce*", "*se*", "*ee*", "*ne*", "*is*", "*p*", "*e*" or "*n*") on each of the four sex/species combinations (male mouse, female mouse, male rat, female rat).

The carcinogenicity classification information was parsed in order to classify the molecule into one of only 2 classes for each specie-sex combination ("*positive*" i.e. carcinogenic and "*negative*" i.e. non-carcinogenic classes). This corresponds with the method used also in the literature (Landwehr, 2010):

- if the molecules of the training set were initially marked "*ce*", "*se*" or "*p*", they were considered as "*positive*" molecules;
- if the molecules of the training set were initially marked "*ne*" or "*n*", they were considered as "*negative*" molecules;
- remaining classes marked "*ee*", "*e*" or "*is*" were considered as "*equivocal*" and as such were excluded from the data sets;

For some compounds, the different classification was obtained for different animal specie or sex, for some compounds a test on the specie may be also missing. Therefore, the total numbers of positive and negative compounds differ according to the specie- sex combination. The Table 7.2 below summarises basic information about the *PTC* dataset.

Table 7.2 – The *PTC* datasets, basic parameters

Number of predicates	atom/3	10 703 occurrences
	bond/4	10 873 occurrences
	other predicates	41 different predicates, 6 497 occurrences in total
Number of examples in classes	Male mouse (*PTC* MM)	129 positive, 207 negative
	Female mouse (*PTC* FM)	143 positive, 206 negative
	Male rat (*PTC* MR)	152 positive, 192 negative
	Female rat (*PTC* FR)	121 positive, 230 negative

Table 7.3 – The *CAD* dataset, basic parameters

Number of predicates	Total structural predicates	16 674 occurrences
Number of examples	Positive examples	55 examples
	Negative examples	41 examples

CAD dataset. The *CAD* dataset represents a real-world engineering task. This problem describes the domain of *CAD* documents (product structures) (Žáková et al., 2007) and the dataset consists of 96 class-labeled examples each of which describes one mechanical parts described by a *CAD* document. Specificity of the dataset and the reason why it was chosen to our experiments is that longer features are needed to obtain reasonable classification accuracy. Each of the examples contains several hundreds of first-order literals. The Table 7.3 below shows brief summary of the dataset and gives brief information about number of the background predicates that were used for clause construction within the *CAD* domain. Part of the semantic structure that was used to describe the work pieces is shown below in Figure 7.1.

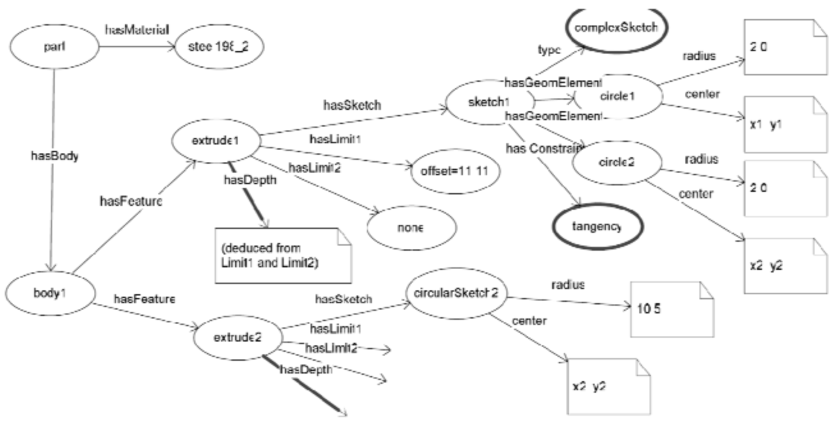


Figure 7.1 – Part of a semantic annotation used for description of the CAD schemes in the *CAD dataset* (Žáková et al., 2007)

Alzheimer problem. This problem also known as *Tacrine drug* includes four benchmark data sets previously used to evaluate a variety of ILP and relational learning algorithms. The task is to predict the relative biochemical activity of variants of Tacrine, a drug for Alzheimer's disease (King et al., 1995). 37 variants of the drug are described in the data – each compound was created by substituting one or more chemical substructures in the original structure of the Tacrine compound.

Table 7.4 – Summary statistics for Alzheimer's drug data sets

Dataset	Alzheimer acetyl	Alzheimer amine	Alzheimer memory	Alzheimer toxic
Number of examples	1 326	686	642	886
Majority class	50%			
Number of predicates	30 predicates, total 3 754 facts in background knowledge			
Property followed	high acetyl cholinesterase inhibition	inhibition of amine re-uptake	good reversal of scopolamine-induced memory impairment	low toxicity

In contrast to the previous problems, this problem is not a true classification task but represents relation learning task. In each of the four datasets, one key property of the drug needs to be compared: low toxicity, high acetyl cholinesterase inhibition, good reversal of scopolamine-induced memory impairment, and inhibition of amine re-uptake. For each property, the positive and negative example datasets contain pair wise comparisons of drugs (however, not for all pairs of the drug variants the relation is known). These compounds are ordered in pairs ($c1$, $c2$) of each labeled as positive if and

only if $c1$ is rated higher than $c2$ with regard to the property of interest. The goal of the problem is following: having two drug variants the system should predict which of these two has higher value of the predicted parameter. For example, '*less toxic(c1, c2)*' means that toxicity of the drug $c1$ is lower than that of $c2$.
The background knowledge data contains facts about the physical and chemical properties of the described Tacrine variants (e.g. polarity or hydrophobicity) and the relations between the physical and chemical constants used.
Because the datasets within each problem type are similar and the experiments would show similar results, we do not run the experiments for all of the datasets but select one dataset for each of the problem type. For this purpose we selected *PTC MM* task to represent the *PTC* problem and *Alzheimer memory* to represent the *Alzheimer* problem. All datasets are used only in the Section 7.5 where we present benchmarking comparison of results of our system to other algorithms.

7.1.2 The Actions

The set of actions that was used in the experiments is given in Table 7.5 (see Table 6.2 and Table 6.3 for detailed description of the actions and their function). Except for the basic refinement actions we use the "*less than*" and "*greater than*" actions for the *PTE* dataset as only this dataset contains numeric parameters (real numbers).

Table 7.5 – Actions used in the experiments

Action group	Actions used
Not context sensitive actions	addLiteral(L), removeLiteral(L), deleteConstant(C)
Context sensitive actions	unifyVariables(V1, V2), instantiateVariables(L,A,C), addLiteral(L,A,N), addNonEq(V1,V2)**, addLTE(L,A,C)*, addGTE(L,A,C)*,

* *Action used only with PTE datasets*

** *Action used only with PTE and PTC datasets*

Additionally, for the chemical datasets *PTC* and *PTE* we use the "*non-equality*" action that adds the non-equal relation ($\neq$) to the clause. This was motivated by the fact that using only the literals from the language used in the data to describe the molecular structures does not allow for sufficient expressivity. For example, without the non-equality it is not possible to construct a clause that would distinguish between a pair and a chain of atoms of same type (illustrated in Figure 7.2 below).

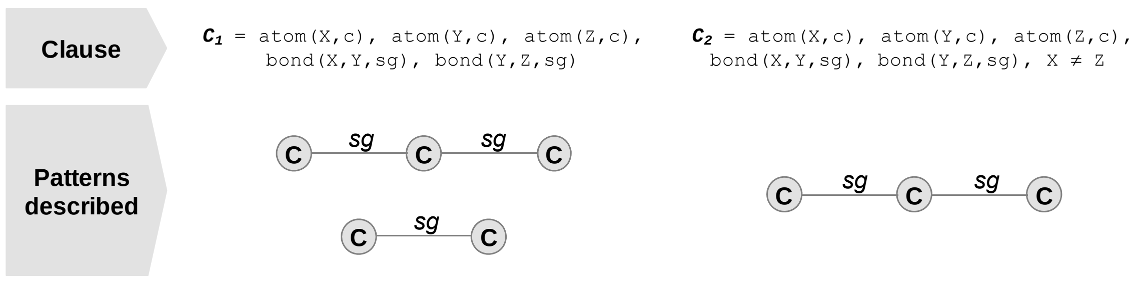

Clause
C1 = atom(X,c), atom(Y,c), atom(Z,c), bond(X,Y,sg), bond(Y,Z,sg)
C2 = atom(X,c), atom(Y,c), atom(Z,c), bond(X,Y,sg), bond(Y,Z,sg), X ≠ Z
Patterns described
C
sg

Figure 7.2 – Ambiguity of described molecular patterns (*PTE* and *PTC* datasets) when not using or using the non-equality

7.1.3 Performance Evaluation Metrics

In the experiments we follow two main criteria - the *predictive accuracy* (quality of the final solution) and *search complexity* (size of the space searched). The reason for using the predictive accuracy and number of constructed clauses as the main performance criteria in this thesis is obvious - we want to tune our system so that it is able to produce the most accurate model while searching reasonable number of clauses. This search complexity is assessed by the number of unique clauses generated during the search because several computers were used to evaluate the function of the algorithm and the computation times would not be comparable. Additional criterion that gives information about the search complexity is the model size expressed as the number of clauses the models contain.

Predictive accuracy. The primary performance evaluation criterion for the experiments in this thesis is the sample classification accuracy measured over the instances that were not used to train the classifier (although others based on information theory have been suggested e.g. in (Cleary et al. 1996).
The sample accuracy represents an estimate of the "*true*" accuracy of the algorithm. It is the probability that the algorithm will correctly classify an instance drawn from example set with unknown distribution of examples. Common practice used also in this work is to split the data into training and test subsets. The model is trained using the training data and tested using the test data partition where the classification accuracy is calculated. Thereby we reach the accuracy estimate calculated as the percentage of test examples correctly classified by the algorithm. Estimating the accuracy on a test set of examples is better than using the training set because examples in the test set have not been used to induce concept descriptions. Using the training set to measure accuracy will typically provide an optimistically biased estimate, especially if the learning algorithm over-fits the training data.
We are using the cross-validation approach where the data is randomly split into k mutually exclusive subsets of approximately equal size. A learning algorithm is trained and tested k times - each time one of the folds is used for testing and the remaining $k-1$ folds is used to train the algorithm. The cross-validation estimate of accuracy is then the overall number of correct classifications divided by the number of all examples in the data. It is common practice we repeat the cross-validation n times in order to provide a stable accuracy estimate (Dietterich, 1988). The selection of examples to the cross-

validation folds was done by stratified sampling to ensure that the class distribution from the whole dataset is preserved in the training and test sets. Motivation for this was that stratification helps reducing the variance of the estimated final accuracy (Kohavi 1995).

Search complexity. Because we run our experiments on several machines we can not use the time needed for the algorithm run as the universal measure. Therefore, we focus rather on the number of clauses generated by our algorithm during the search. To assess the computational requirements this number can be measured in two ways. The options are either count all the clauses generated during the search or measure only the number of unique clauses constructed during the search.
In our case, the second option represents better estimate of the time demand of our algorithm as we buffer the clauses that were once evaluated in the iteration and the lengthy θ-subsumption tests are not run for all clause but only on the new clauses. Charts showing the correlation between number of clauses and time of the algorithm run are given in the appendix (see Figure A.4). Both the scatter plots and the R-square indexes presented in the figure show the dependency between the number of unique clauses and the runtime is much stronger than the dependency between the total number of clauses searched and the runtime. Therefore, we will use the number of unique clauses as an approximation of the search complexity.

Model complexity. Complexity of the final model (average number of clauses in the model) gives information about the behavior of the search algorithm. In some setting the learner might create large model consisting of patterns with small but "pure" coverage exposing the result to the risk of overfitting, in other set-up it might create a model with few clauses missing the necessary detail.
In all the experiments, unless stated otherwise, we present the averages from five batches of five-fold cross validation to assess the parameter. Each dataset is divided in five disjoint sets of similar size, one of these sets forms the test set, and the union of the remaining four the training set. The system is run five times each time with different random seeds to split the examples into five subsets and then uses subsequently each of the sets as test set and learning on the remainder (so the algorithm is run 25 times per each dataset). The outputs (classification models) are evaluated on the corresponding test sets.
In the next sections we will analyze the key parameters of our POEMS-based ILP system on each of the three basic layers. We will start with validation of the concept of context-sensitive actions used on *Layer 3*.

7.2 Context Sensitivity of the Actions

In this section we will analyze the effect and usefulness of using context-free and context-sensitive actions on the POEMS learning process. According to the scheme of the system (see Figure 6.2 and Figure 6.4) the context information is automatically induced prior to start of the learning phase and then used on the *Layer 3* in the process of translation of the numeric arrays into action sequences. This layer represents interface between ILP task defined in FOL and the chromosome based evolutionary searcher. The algorithm takes the input - numeric array – and based on the list of the user-defined refinement actions it translates the numeric array into a sequence of refinement actions. Finally, this sequence is applied on the starting phenotype to see and evaluate the effect of the sequence. In our

approach, the actions are context-sensitive and they use the context information the system learns prior to entering the model construction phase.
The main issue to be proved is whether the utilization of the context significantly improves quality of the search and really helps the POEMS algorithm to produce meaningful clauses (i.e. clauses with non-trivial coverage that cover neither all nor any of given examples). To show the effect of the context we will run two experiments.

1) *Focusing on the useful clauses* - for both options (with and without context) we randomly generate clauses of different complexity and analyze the difference in the coverage of both groups of clauses; the clauses generated with context utilization should show much higher ratio of non-zero covering clauses.
2) *Accuracy of the final model* - for both options we will run the full POEMS algorithm and analyze the accuracy of the final model; the hypothesis to prove is that context enables our algorithm to reach higher accuracy of the final model.

7.2.1 Focusing on the "Useful" Clauses

In the first set of experiments we analyzed the ability of our approach to randomly generate "*useful*" clauses i.e. clauses that cover at least some of given examples avoiding the clauses with zero coverage. This is important for the whole functionality of the stochastic search process, because the GA-driven search is performed through stochastic process that builds on preference and modification of better clauses. Clauses with zero coverage are not suitable as all of them carry the same evaluation function values and it is difficult to prefer some of them based on the clause quality preference calculated from coverages.

7.2.1.1 Structure of the Graph of Relations

Prior to the experiment we have briefly analyzed the graph of relations of each dataset to get picture about how utilization of the graph of relations can help in the refinement-based clause search. To answer this question we looked at the density of the graph - if the graph is highly connected then the difference between using and not using the context should be lower than in the case of more structured dataset. The reason is that we use the context information to limit undesired connection of literals with other literals or constants. The more combinations will be forbidden (will not occur in the data) the higher effect will the utilization of the context have. For example, we expect that the effect for the *PTE* dataset that uses mainly the two predicates that define atoms and bonds will be lower than for the *CAD* dataset that is described by more complex semantic scheme (see Figure 7.1).
We used two measures to get a basic picture about the structure of the datasets – the average node distance and the average node degree. The average node distance is defined as the average number of nodes that need to be passed on the shortest path between two other distant nodes n_1 and n_2. The average node degree expressed as percentage of theoretical maximal node degree shows how many nodes are connected to given node compared to maximal number of possible connections.

Table 7.6 – Properties of the Graph of relations for each dataset (nodes for constants omitted)

Dataset	Average node distance (# edges on shortest path)	Avg. node degree (% of maximal degree)
Trains	1.50	43.0%
CAD	2.01	28.1%
PTC MM	1.11	89.6%
PTE	1.30	49.1%
Alzh. memory	1.91	39.1%

It should be noted that in both cases we do not count theoretical possibility of connection of nodes that represent arguments of the same literal as we do not have this case present in our data (e.g. '*hasCar(X,X)*' is not possible). Graph of relations for *CAD* and Trains datasets are given in Figure 7.3 and Figure 7.4 (graphs of *Alzheimer*, *PTC* and *PTE* datasets are too complex to be presented here due to the quantity of nodes they consist of). The average values for the two indicators for each dataset are given in Table 7.6.
From the results in Table 7.6 we can see that the graph of CAD dataset has both the highest average node distance and lowest average node degree. Each predicate argument can share the same variable on average only with 28% of other predicate arguments. These indicators confirm our expectation that *CAD* data is highly structured and the context utilization should have the highest effect.

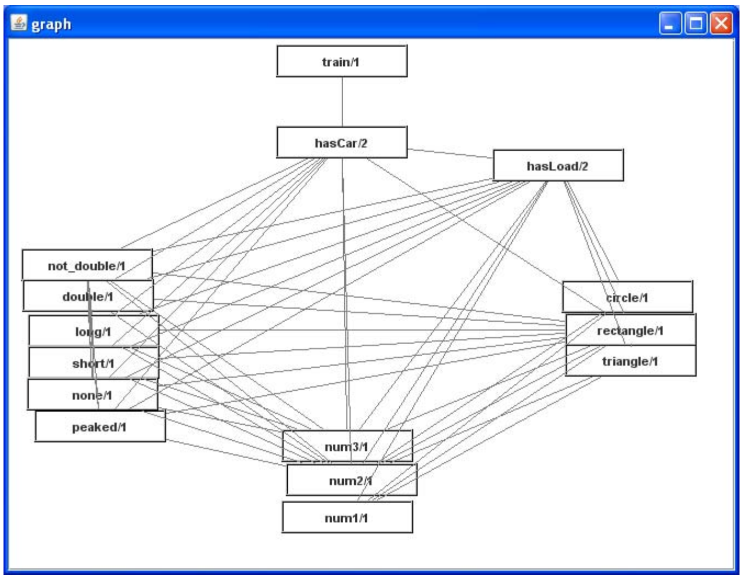


Figure 7.3 – Graph of relations for the *Trains* dataset (nodes of arguments of the same predicate merged together for better transparency)

On the other hand, according to the indicators the graph of relations for the *PTC* dataset is highly interconnected – the average node degree shows that a predicate argument can share variable with almost 90% of other arguments. Also the average node distance is very low showing again high connectivity of the graph – the number 1.11 indicates that on average 9 out of 10 predicate arguments are directly connected meaning that they can share the same variable. Based on this we hypothesize that the effect of context utilization should be lowest of all the four datasets (the effect can be seen e.g. in the comparison of coverage of clauses generated fully at random and clauses generated with context utilization).

7.2.1.2 *Generating Random Clauses With and Without Context*

To analyze the effect of context on the ability to generate "*useful*" clauses we carried out following experiment: for each dataset, we defined seven groups of clause complexity (= number of literals + number of constants in the clause) and for each of these groups we generated 1000 clauses by starting with empty clause and randomly applying clause refinement actions with random parameters until the clause reaches complexity needed. Finally we evaluate coverage of the clause. The clause coverage can be of one of three types: zero coverage, full coverage (covers all examples) and the remaining clauses that have the non-trivial coverage (clause covers neither all nor zero examples). Last two cases we can label as "*useful*" because such clauses can be further evolved to reach some well classifying concept.

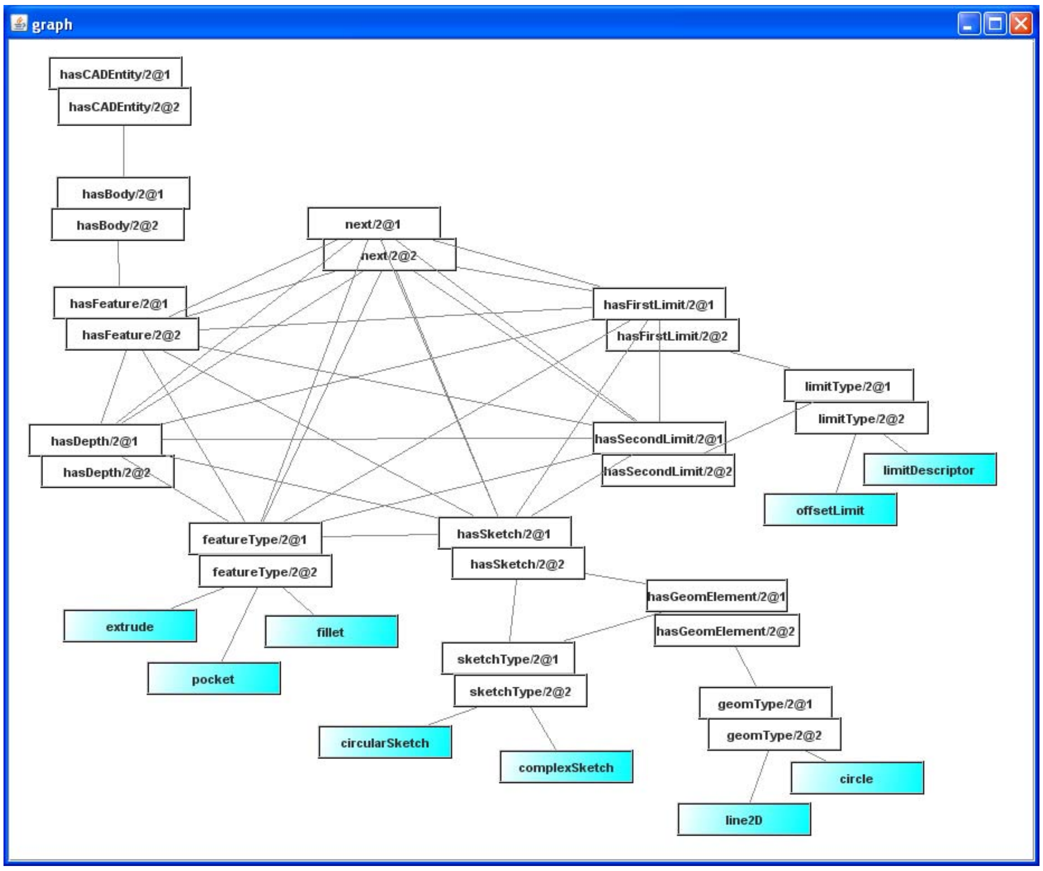


Figure 7.4 – Graph of relations for the *CAD* dataset (nodes representing constants in blue, nodes of arguments of the same predicate put next to each other for better transparency)

We run this experiment twice – first with context information (the set of potential action outcomes was constrained based on known context relations) and then without using the context information (i.e. for each context-sensitive action no filtering was used and all action outcomes were possible in each application).

The results of this experiment for the *Trains* and *CAD* datasets are graphically summarized in Figure 7.5, results for *PTE* and *PTC MM* datasets follow in Figure 7.9 and the comparison of clauses generated with and without context for the *Alzheimer memory* dataset are in Figure 7.10.

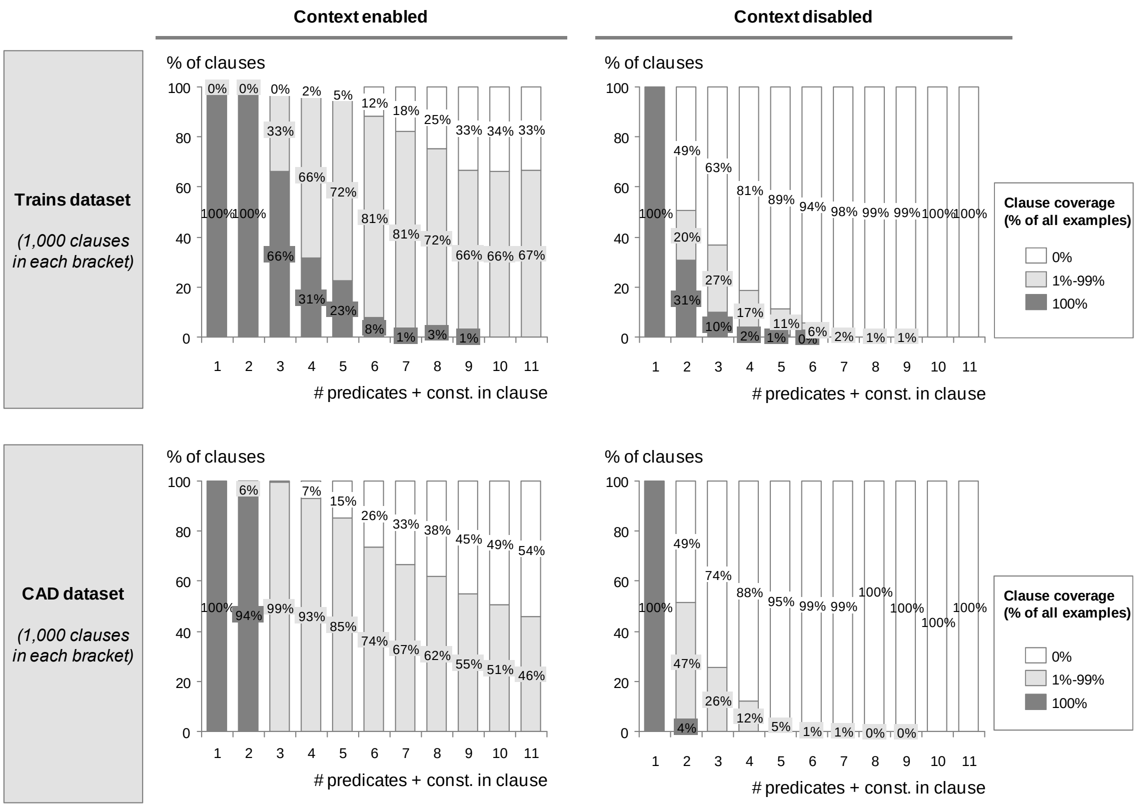
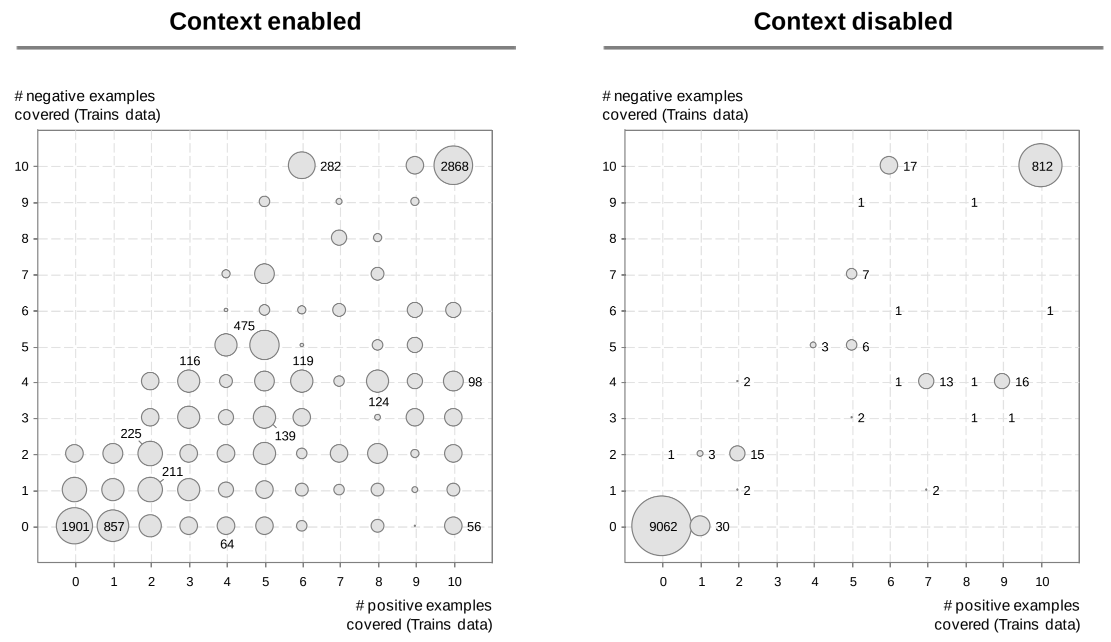


Figure 7.5 – Random generation of clauses of different lengths (1,000 clauses in each clause complexity bracket) with and without enabling context sensitivity (*Trains* and *CAD* dataset)

Trains dataset. The *Trains* dataset is artificially generated and each example can be depicted as a tree-like structure. The previous analysis showed that the connectivity of the arguments in the Graph of relations of this dataset is around 40%-50%. This is reflected also in the comparison. Majority of the clauses generated with context-sensitive actions have non-zero coverage regardless of the clause complexity (even for the length of nine predicates the portion of zero covering clauses is only 33%). On the contrary, the coverage of clauses generated without context sensitivity of the refinements decreases fast with growing clause complexity the chance to generate meaningful clauses longer than 7 predicates (which is also the length of the discriminating concept) is below 2%. The context sensitivity of actions appears to be necessary for successful problem solution in this domain by means of our refinement based stochastic search.

This conclusion is supported also by the scatter plot in Figure 7.6 that shows the distribution of randomly generated clauses in the space of clause coverages. This plot was created to give us a brief idea about what kinds of clauses are generated by the algorithm regardless of their size but with respect to their coverage. In the experiment that is summarized in the figures we generated 10,000 clauses with action sequences of length 2 to 12 actions and plotted the coverage of the clauses and their quantity. Results for Figure 7.6 a) were generated with using the context information, Figure 7.6 b) shows the results of random clause generation without using the context.



a) coverage of randomly generated clauses with action context sensitivity enabled

b) coverage of randomly generated clauses without action context sensitivity (i.e. no context used)

Figure 7.6 – Distribution of 10,000 randomly generated clauses in the space of clause covered examples (*Trains* dataset, all data)

The difference in this comparison is obvious – in the case when no context information was used, 91% of clauses (9,062 clauses) cover no example, 8% clauses (812 clauses) cover all examples and only 1% of clauses have non-trivial coverage. In the case when context was used, only 19% of clauses cover no example and 29% clauses cover all examples. Remaining 52% of clauses have non-trivial coverage and we can see they are distributed across a large part of the space. Additionally, 56 clauses (~0.5%) have ideal coverage of 10 positive and no negative example.

***CAD* dataset.** Comparison of clauses generated with and without context shown in Figure 7.5 confirms our expectations about the effect of context in the *CAD* dataset - as expected from the brief analysis of the graph of relations connectivity (to view the graph see Figure 7.4), for the *CAD* dataset the context utilization has significant effect. While the majority of the clauses generated randomly without context utilization shows zero coverage from the clause size of 3 (see right hand side of Figure

7.5), the coverage of clauses produced with the context shows different characteristics. Up to the size of 5 predicates/constants almost all the clauses has full or non-trivial coverage. In the groups of more complex clauses we can see also clauses with zero coverage, however the share remains below 54% (for the size of 11).
Additional information that we can see from the chart is confirmation of our expectation that longer clauses are needed in *CAD* problem to build discriminative theory as large share of the shorter sequences has full coverage.

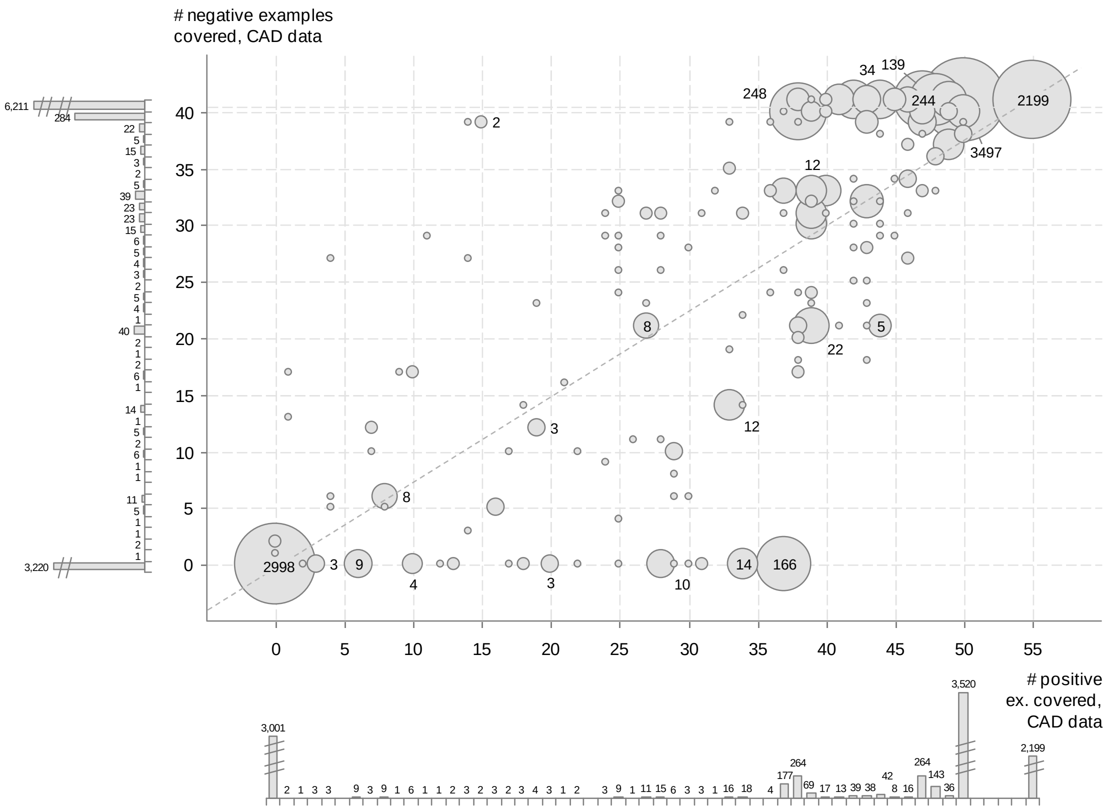


Figure 7.7 – Distribution of 10,000 clauses randomly generated with enabled action context sensitivity in the space of coverages (CAD dataset, all data)

Like for the Trains dataset, also for CAD we performed the analysis of distribution of the randomly generated clauses as to the coverage these clauses have. The plots of distribution of 10,000 randomly generated clauses according to the number of positive and negative examples the clauses cover are depicted in Figure 7.7 (clauses generated with context sensitive actions) and in Figure 7.8 (the clauses generated without context). The chart in Figure 7.7 shows that only 30% of clauses (2998 clauses) generated with context had zero coverage while the remainder showed "*useful*" coverage. 22% of the clauses (2199 clauses) had full coverage and another 35% of generated clauses (3497 clauses) covered all but 5 positive examples. The remaining 13% of clauses showed non-trivial coverage and were mainly located in the right hand corner of the chart meaning their coverage is rather high. However, over 2% of clauses (222 clauses) show "ideal" coverage as they cover only positive examples without covering any negative one (located on the bottom line of the scatter plot).

The chart shows also another interesting fact – 166 clauses out of the 10,000 (~1.7%) had the coverage of 37 positive examples and no negative example. Using this clause as the only one in the classification model yields the classification accuracy 81.3% (as we make an error on the remaining 18 out of 55 positive examples). This actually seems to be the accuracy presented in the original work (Žáková 2007) that introduced this data (see Section 7.5 for the comparison). We can see that with the context sensitive refinements, 2 out of 100 random efforts should reach this accuracy as well.

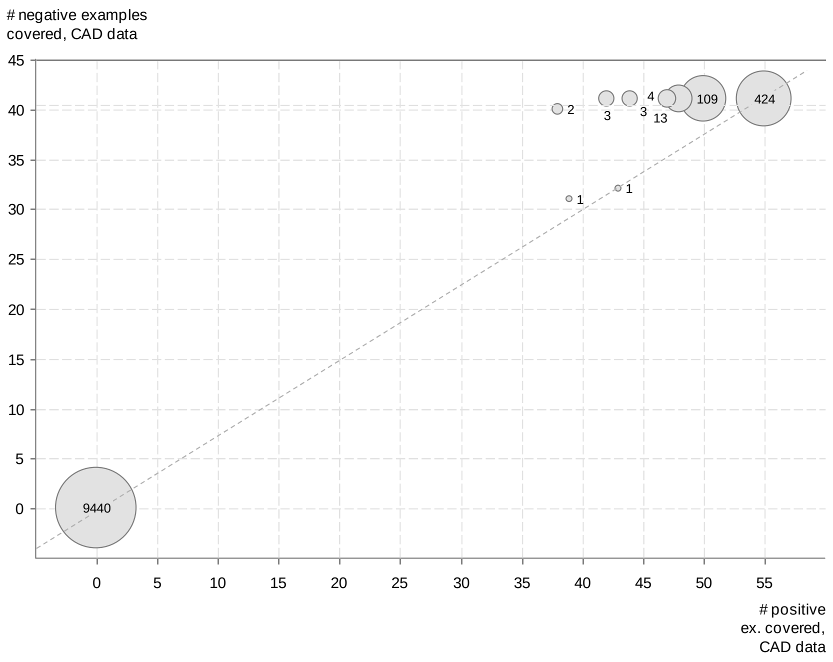


Figure 7.8 – Distribution of 10,000 clauses randomly generated without action context sensitivity in the space of coverages (CAD dataset, all data)

The scatter showing this distribution of clauses produced for the *CAD* problem without enabling the context sensitivity is in Figure 7.8. The comparison in this case requires no comment as there is almost no other clause than with either zero coverage (94% of clauses) or full coverage (4% of clauses).

***PTE* and *PTC MM* datasets.** Comparing the results of next two datasets (see Figure 7.9) we can see that the chemical datasets (*PTE* and *PTC MM*) differ from the previous two. In contrast to the *Trains* and *CAD* problems, in the case of these datasets it is possible to generate meaningful shorter clauses even without the context sensitivity. The reason lies in the structure of the data and was also indicated in the connectivity analysis – the *PTC MM* and *PTE* datasets have simpler structure where almost any literal can be bound by variable directly to any other predicate. This is hardly possible in the more structured *CAD* and *Trains* datasets.

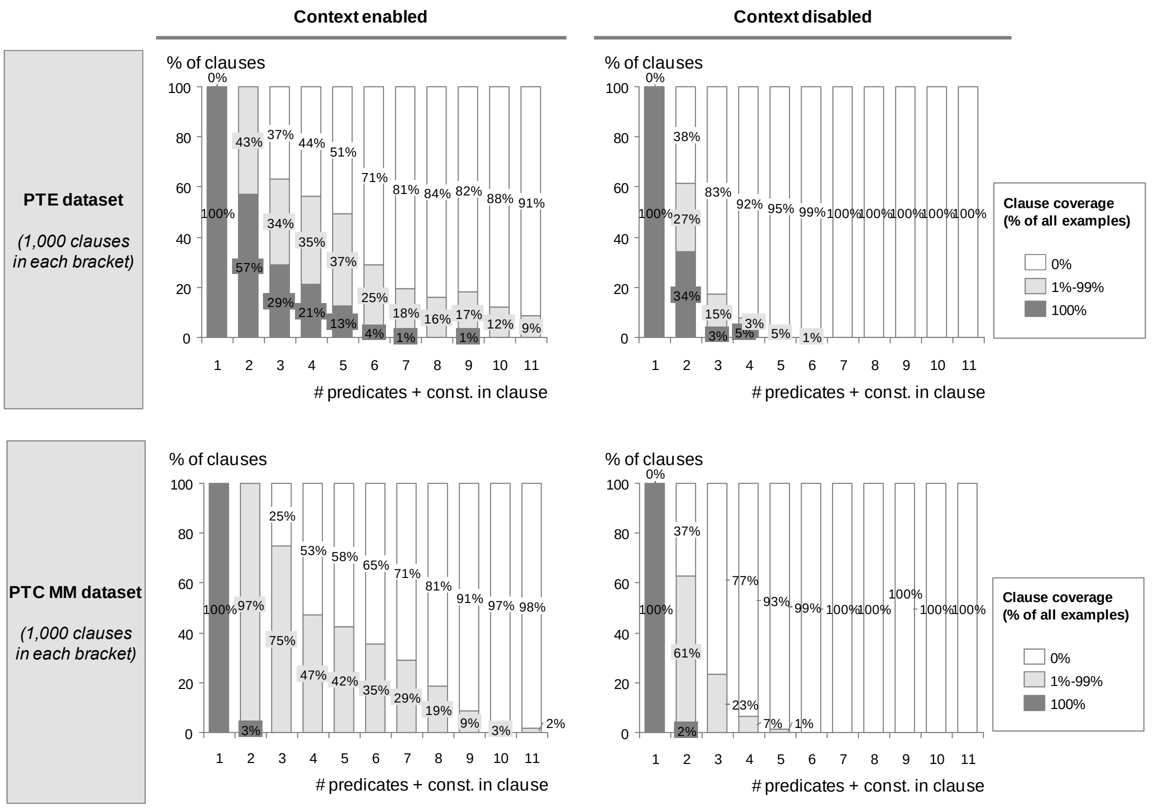


Figure 7.9 – Random generation of clauses of different lengths (10,000 clauses in each clause complexity bracket) with and without enabling context sensitivity option (PTC_{MM} and *PTE*)

This effect is projected also to the results of the experiment shown in Figure 7.9. The effect of using context is lower than in the case of *CAD* data as we can see reasonable share of non-zero covering clauses generated also without the context. Despite that, the difference between context sensitive and non-sensitive clause generation still remains significant – the graphs on the right hand side of the chart show that without context it is almost impossible to generate meaningful clauses of complexity 5 and higher.

Comparison of coverages of clauses generated with and without context for the *Alzheimer memory* dataset is in Figure 7.10. Also here is the difference between using and not using the context information significant. Without context, there are 73% of zero covering clauses already in the group of clauses with two predicates/constants. For more complex clauses this share reaches to 94% and higher which means that longer randomly generated clauses are useless for the purpose of our stochastic search. On the contrary, when context is utilized the share of zero-covering clauses in the group of clauses with 2 predicates/constants is 0%. Even for clauses with 6 predicates/constants the share of clauses with zero coverage is only 45% meaning that more every second randomly generated clause of this complexity is "*useful*".

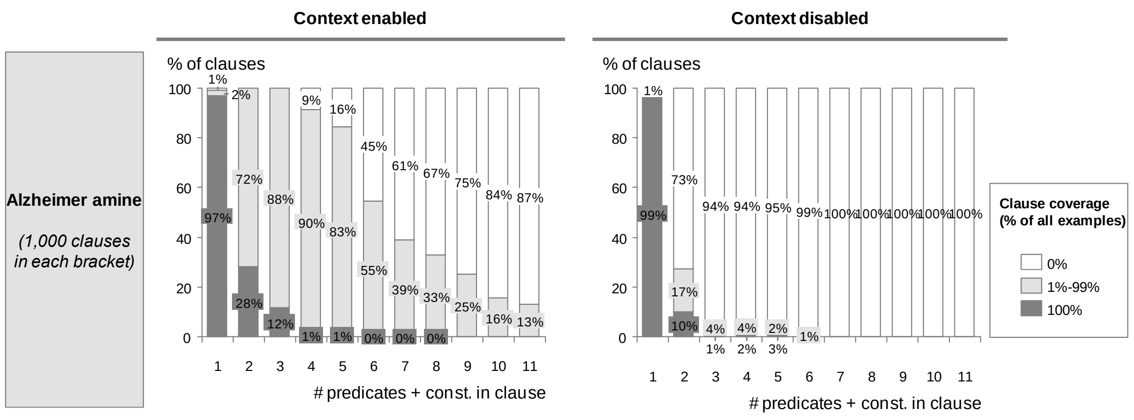


Figure 7.10 – Random generation of clauses of different lengths (10,000 clauses in each clause complexity bracket) with and without enabling context sensitivity (*Alzh. memory*)

From this experiment with random clause generation it is clear that utilization of context is definitely meaningful as it limits the search in the part of clause space where the clauses cover no examples. The effect is higher with the datasets that are more structured, i.e. where the share of arguments that can share the same constant or variable is lower like in the *CAD* dataset.

7.2.1.3 Searching Among the "Useful" Clauses - Conclusion

The main conclusion of the presented comparisons clearly is that the context-sensitivity of refinement actions has strongly positive effect on the quality of generated random clauses. Although depending on the dataset, for all the datasets the share of clauses that have zero coverage was significantly lower when the context information was used and the difference increased strongly for more complex clauses.
The effect is more significant in datasets with "more structured" semantics – the *Trains* and *CAD* data for which the ratio of clauses with non-zero coverage is around 50% even among clauses of complexity 15 literals + constants. By utilization of our context-sensitive refinement concept we are able to generate at random even long clauses that are still meaningful. This stands in strong contrast to situation when context is not used as the share of clauses with at least 3 predicates/literals that have non-zero coverage generated in that case is close to zero.
The experiments also show that combination of random numeric array generator with the context-sensitive refinement actions can serve as efficient (and even stand-alone) random clause generator. Apart from the charts presented in this section, further evidence of this can be seen in Figure 7.20 and Figure 7.22 in the next chapter - the charts show that for all the ILP problems (depending on the selected sequence length) the share of unique non-zero covering clauses can reach up to more than 40% of the total number of randomly generated clauses (and can be even higher for *CAD* and *Trains* datasets).
In the next sub-section we will analyze the effect of context utilization on the accuracy of the models produced by our POEMS ILP system.

7.2.2 Accuracy of the Model with and without Context

To definitely confirm our expectation that context sensitivity is an important feature of our POEMS-based ILP searcher we have used POEMS to build full predictive models with and without context utilization for the *CAD*, *PTE*, *PTC-MM* and *Alzheimer memory* datasets. We used the set-up of the algorithm according to the parameters given in Table 7.9 and run five times the five-fold cross-validation experiment.

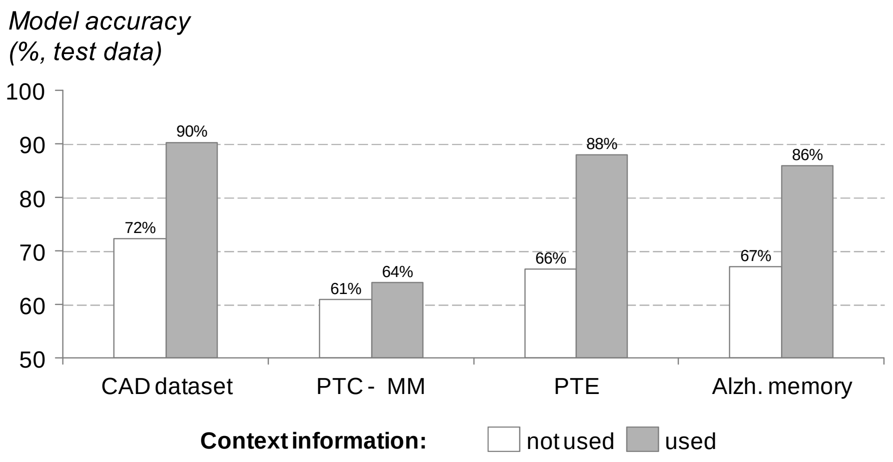


Figure 7.11 – Comparison of the predictive accuracy of models that were built with and without context utilization (test data)

The predictive accuracy results measured on the test data (average from the five-fold cross validation) are summarized in Figure 7.11. The chart proves importance of context utilization as the average predictive accuracy of models built without the context option is significantly lower: the difference varies between 22% for *PTE* to 3% for *PTC MM* datasets.

7.2.3 Context-sensitivity – Conclusions

Basic experiments with the context-sensitivity proved that the context information represented in the form of *Graph of relations* gives the stochastic searcher enough power to create meaningful clauses by random. Both experiments show that context-sensitiveness plays an important role in our application. Without utilization of the context sensitivity the blind stochastic application of the refinement actions does not lead to useful results because majority of the generated clauses has zero coverage. On the other hand, the experiments proved that when the context is used, our system is capable of generating large amount of meaningful clauses at random which is a property needed by any ILP stochastic search algorithm. Except that this allows our EA-based searcher to produce good solutions in reasonable time we saw that the combination of random generator of numeric arrays with context-sensitive actions can be actually used as an efficient standalone random clause generator (by efficient we mean that the generator is able to produce complex clauses with several literals and with non-trivial coverage in the same time).

7.3 The Action Sequence Optimization

In this section we will analyze four most important parameters that influence the action sequence optimization process performed on the *Layer 2* of our algorithm. On this layer the system runs the search for the clauses (see Figure 6.2 and Figure 6.5) – the system iteratively runs the optimization algorithm that searches for the best refinement *meta-operator* made up of a sequence of refinement actions (coded in the form of byte arrays) that modifies the starting prototype clause in the best possible way. We will analyze these four parameters (see Table 7.7 for summary):

1) *Sequence optimization algorithm.* Our POEMS system can utilize for the internal sequence optimization any of the optimization algorithms that are built to work with numeric array representation. As was described in the section on ILP, the ILP tasks are very hard and our effort in this work is to focus on using stochastic Evolutionary Algorithms to solve the task. Yet, even in among EAs there is still a variety of choices. In this work we do aim to examine all possible algorithms to be used with POEMS for ILP. We will briefly test POEMS for ILP with following stochastic optimizers: basic Monte Carlo (MC) random searcher (MC), standard Genetic Algorithm (GA), Evolution Strategies (ES) and Simulated Annealing (SA) to select the proper algorithm to use. The main criterion will be the predictive accuracy (average over test data in cross-validation).

2) *Evaluation function.* Although the main quality criterion for the whole model is the predictive classification accuracy, the search for partial clauses/models does not necessarily have to follow the same criterion. The standard approach taken by most of the ILP systems is actually to use different criterion than accuracy to guide the search in the space of clauses (or theories) – this holds both standard approaches (e.g. Aleph or Progol) and EA-based approaches (e.g. ECL system).

   We will test six basic evaluation functions with the aim to select the best one as to the predictive accuracy of the resulting model (average over test data in cross-validation). Apart from this, we will analyze the number of clauses searched and average size of the model to have an overview about the effect of the evaluation function on the search efficiency.

3) *Search strategy.* Our system is designed to iteratively construct the refinement path to the final solution represented as *meta-operator* that changes the starting clause to clause with good coverage for classification. The basic question is whether the length of the optimized action sequence should be set so that it is possible to construct this *meta-operator* in one step or it should be set with the assumption that several iterations are needed to reach the optimal solution. The first option means searching in a larger space of refinement action combinations, the second option searches smaller space but exposes the learner to local optima which can disable reaching the optimal results. Additionally, utilization of too long sequences can lead not only to reaching the result in longer time, but also to overfitting or potentially even missing the optimal clause (if the clause is of a lower complexity and can be "jumped over").

   We need to determine the optimal combination of the sequence length and number of iterations that would enable reaching the balance between runtime and quality of the final model so that good results can be produced in reasonable time. To do this, we will run the system with different threshold on the sequence length and on the number of iterations and compare the results (accuracy of the resulting model, number of clauses searched).

4) *Generalization of discovered clause.* Motivated by decreasing the risk of overfitting that may be caused by over-specializing the refinement sequence we included a special step to our POEMS-based system after the clause is discovered – clause generalization. The open question to prove is whether this step improves the system learning ability and quality of the output (the criterion being predictive accuracy of the model).

Table 7.7 – Summary of open questions regarding sequence optimization layer () targeted in this section

Issue	Question	Options	Criteria
Optimization algorithm	Select the most efficient optimizer for POEMS	GA/SA/ES/MC	Predictive accuracy, size of the space searched (measured in number of unique clauses generated)
Evaluation function	Select proper evaluation function to guide the search	Six evaluation functions	Predictive accuracy, size of the space searched (measured in number of unique clauses generated)
Search strategy	Find the optimal balance between the number of iterations and size of the individual action sequences	More iterations with short sequences or vice versa.	Predictive accuracy, size of the space searched (measured in number of unique clauses generated)
Overfitting	Test the contribution of generalization prior to adding clause to the model	Yes/no	Predictive accuracy

7.3.1 Selecting the Optimization Algorithm

In this section we try to briefly confirm our hypothesis that genetic algorithm is a proper option of the optimization algorithm that is used to search for the best refinement sequence in our POEMS system.
Because the POEMS algorithm allows to easily use any of the optimization algorithm that operate with numeric arrays we have tested four basic stochastic optimization algorithms to see whether the results can be better with other optimizer: standard genetic algorithm (GA), simulated annealing (SA) and evolution strategies (ES). The reason for selecting the SA and ES is the frequent motivation for selecting these algorithms also in other cases – they achieve good results while the set-up of these algorithms is simpler than GA as they take lower number of input parameters.
In the first experiment, we used the *Trains* dataset and compared POEMS using each of these optimizers to GSAT algorithm. We used the stochastic GSAT searcher implemented in Aleph system (Srinivasan 1993). This implementation uses concept of bottom clauses (Srinivasan 1993) used by

Aleph and as such is not completely blind random search. The question was how efficient is each of the algorithms in reaching the ideal concept (how many clauses are searched).

The experiment was carried out for 10 runs and the maximal number of iterations was set to 5 so that in combination with the set-up of the optimization algorithms it is given that the searchers will search at maximum 10,000 clauses. The length of the optimized refinement action sequence was set to 8 actions (as we know the ideal concept contains 7 literals). Each approach started with an empty clause. Because of the limit 10,000 clauses we set up the optimizers in the following way:

– *POEMS-ES* – POEMS with a variant of mutation-based Evolution Strategies (ES) (Kubalik and Faigl, 2006) that can be seen as simplified version of standard GA. It maintains one single actual prototype clause (individual) from which, in each iteration (generation), λ new clauses are created by hyper-mutations. If the best individual out of the λ new individuals is at least as good as the current prototype then it becomes prototype for the next generation and the current prototype clause is discarded. Otherwise, the current prototype remains also for the next generation. The parameter λ was set to 1,000.
– *POEMS-SA* – POEMS with simulated annealing (SA). Starting with a randomly generated action sequence coded as an array of numbers of fixed length, this sequence is optimized by SA process – see e.g. (Serrurier et al. 2004). We used start temperature 100, end temp. 0.1, cooling factor 0.98.
– *POEMS-GA* – POEMS with standard genetic algorithm (GA) as introduced in the text above; tournament selection (best 2 of 4), one-point crossover, random byte mutation, population size: 40, number of generations: 25;
– *POEMS-MC* – POEMS with simple random array generator; the generator was set to select best out of 1,000 randomly generated sequences;

Results of the experiment are summarized in the Table 7.8. The table shows that at 40% of the runs the Aleph GSAT algorithm did not find the optimal solution before reaching the limit of 10.000 clauses explored. The POEMS-based searchers always discovered the solution before reaching the limit. Comparing the individual optimization algorithms in POEMS, POEMS-GA reached the result with lowest number of searched clauses.

Although our approach does not necessarily require GA to be used as the optimizer, this is one of the reasons why we consider the GA to be suitable optimizer for solving other problems and use it further for other comparisons.

Table 7.8 – *Trains* problem, number of clauses constructed before 100% accuracy reached

Algorithm type	**Avg. number of clauses constructed**	**Std. dev.**	**Successful runs[%]**(*
Aleph-GSAT	6668.0	4797.1	40%
POEMS-MC	1478.2	829.3	100%
POEMS-ES	779.2	683.1	100%
POEMS-SA	847.8	511.7	100%
POEMS-GA	**447.6**	**496.1**	**100%**

(* Number of runs terminated before reaching limit 10.000 clauses, avg. from 10 runs

In the second experiment we used our remaining ILP datasets to compare the four POEMS variants as to the predictive accuracy of the final model on the test data of the five-fold cross-validation.
The set-up of the individual algorithms was fine-tuned in several (~10) runs of the system on each respective dataset. The parameter setting algorithm runs were performed only on the training set with the main criterion being the best classification accuracy (the test set was not used during these runs).

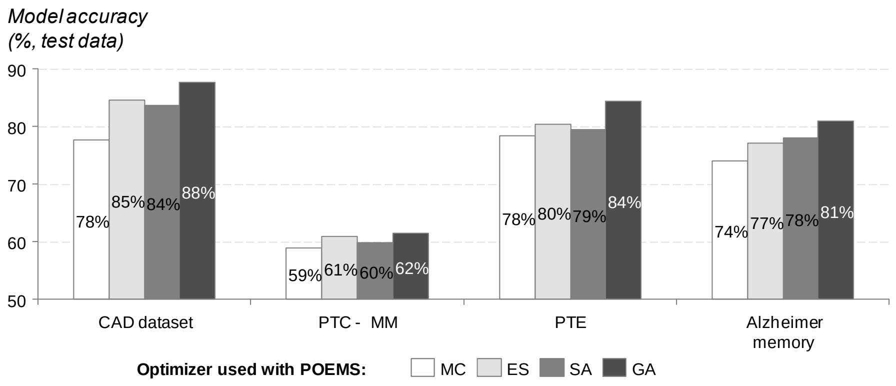


Figure 7.12 –Comparison of the classification accuracy of POEMS with different sequence optimizers (five-fold cross-validation, test data)

The results depicted in Figure 7.12 show that GA outperforms the other algorithms for all the datasets (despite that for the *PTC-MM* data the results for the different optimizers are almost the same). Considering the results of both experiments presented in this section in which the POEMS system with GA gave the best results we will use the GA algorithm in our POEMS-based ILP system. The set-up of the GA we are using in the experiments is summarized in the Table 7.9 below. For more details on the GA used in this work see Section 0 in the Appendix.

Table 7.9 – Generic POEMS-GA algorithm set-up used for experiments

Parameter	Value
Crossover operator	Standard 2-point crossover, probability of crossover $p_{xOver} = 0.8$
Mutation operator	Standard mutation operator, Prob. of byte replacement $p_{Mut} = 0.1$
Selection operator	Tournament selection, best 2 out of 4 competitors
Generations	50 generations
Population size	50 individuals

7.3.2 The Evaluation Function

In the first part of this section we will analyze the effect of different evaluation functions on performance of our system and will select the most proper function. In the second part we will validate the contribution of the solution that we proposed to face the plateau effect – adjustment of the evaluation function that prefers clause complexity.

7.3.2.1 Selecting the Evaluation Function

The evaluation function actually gives shape to the search space by assigning quality level to clauses based on their coverage. As we use POEMS to create the model by set-covering we think that not every function is suitable for this approach. *Our hypothesis regarding the evaluation function is that the predictive performance of models built with POEMS algorithm can be improved if we select proper evaluations function that fits to the characteristics of our algorithm.*

The main quality criterion in assessing the effect of the evaluation function is the predictive accuracy of the final model measured as the ratio of correctly classified examples to all examples. However, using accuracy also for the partial clause construction might not be necessarily the best option. Actually, the approach of using different function for the clause search was adopted by most of the ILP systems – both the “standard” systems like Progol or Foil (see Chapter 3.5) and also those using evolutionary techniques (see Chapter 4.3.3).

In this work we have selected from six following criterion functions: accuracy, entropy (basic and normalized), F-metric, Foil information gain, Progol compression and novelty. For more details on the functions see Section 3.4.1.2. The beta parameter for the F-metric function was set to 0.1 – this setting gave the best results over the training data for most of the datasets.

For the selection of the function we have designed similar experiment to the one realized in the previous section. For each of the functions we run our system on each of the datasets and analyze the basic parameters of the model and the system run – accuracy of the model, number of clauses searched and average size of resulting models. We run the learning process five times each time performing the five-fold cross-validation and present the average values.

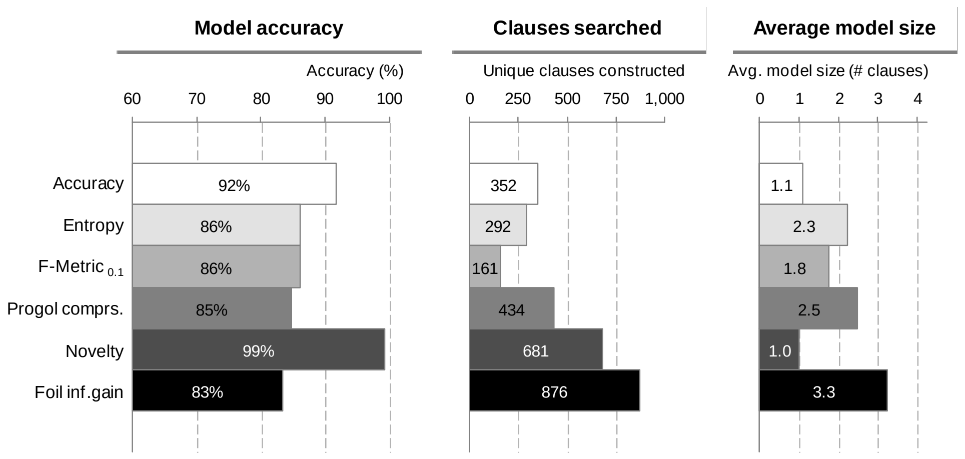


Figure 7.13 – Effect of different evaluation functions on learning output, Trains dataset (all data, 25 runs average)

The first dataset we analyzed was the *Trains* dataset. Results of the comparison of accuracy (all data), average model size and average number of clauses searched are depicted in Figure 7.14.
According to the figure, the best results were obtained with the *novelty* and *accuracy* evaluation functions – the ideal clause was found in almost all of the cases. The other functions show slightly worse results with the Foil information gain, Progol compression function, entropy and F-metric (with the beta parameter set to 0.1) achieving the worst results. The average sizes of the produced models (average number of clauses) show that all these functions lead the searcher to produce too specific clauses. Therefore, the model is formed by 2 or more clauses instead of single clause. On the other hand, these functions search fewer clauses than the novelty function.
The results of this experiment using other four datasets are shown in Figure 7.14. Also here for the *CAD* dataset the average best predictive accuracy is achieved when using the novelty function (both on training and testing data). The entropy, F-metric$_{0.1}$ and Foil information gain show worse results yet with higher average model size. However, for the remaining three datasets (*PTE*, *PTC MM* and *Alzheimer memory*) the best results are achieved with the entropy function followed by F-metric and information gain. On all the datasets, these functions also lead the system to produce models with higher number of clauses than the *novelty* and *accuracy* functions.

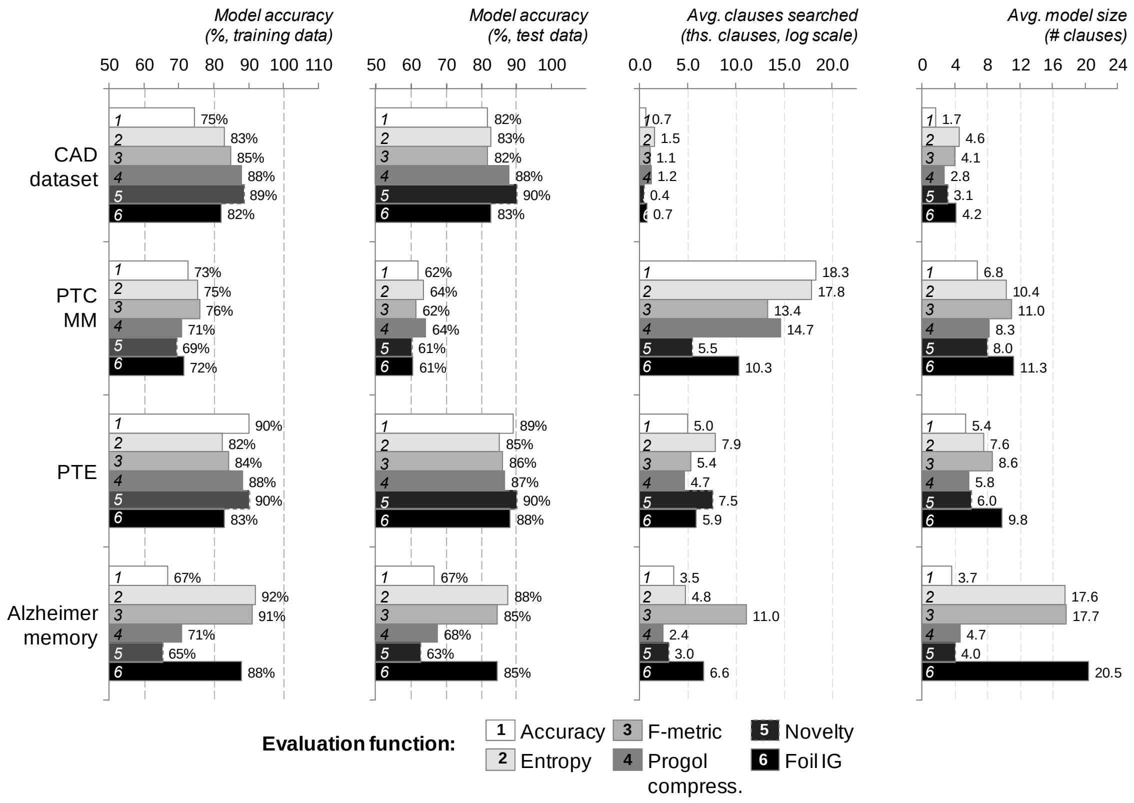


Figure 7.14 – Effect of different evaluation functions on the parameters of the final model

After comparison of the results reached over all the datasets we can see one strong difference between the functions when they are used to guide the set-covering-based learner – the functions like entropy or information gain lead to models built of more clauses each of which of having smaller coverage with covering high share of examples from one of the two classes. The second group of functions like novelty leads the set-covering algorithm to produce smaller models with clauses covering higher number of train examples.

The final effect depends on the character of the analyzed dataset. For the datasets that offer patterns with large coverage like the *CAD* or *Trains* dataset the evaluation function like *novelty* seem to be more proper. This is because this function prefers clauses with higher coverages and does not allow the system to dive immediately to the space of clauses with small coverage.

Different situation was observed for the datasets with higher noise like *PTC MM* or *Alzheimer memory*, Due to the noise present, the datasets cannot be easily divided by using a few patterns and more granular split needs to be applied by the classification model. For this purpose, the second group of functions (e.g. *entropy*) proved to give better results as it tends to produce bigger models with more specific clauses.

7.3.2.2 *The Plateau Phenomenon and Overfitting*

This subsection aims at our solution of the the plateau problem - situations, when the evaluation function is constant in larger parts of the space and the search in these areas goes blind.

The POEMS algorithm in its substance adopts the "standard" solutions to plateau issue (see Section 6.3.3.2). However, problems that are characterized by larger plateaus and require larger classification concepts (like the CAD dataset) still seem to require special treatment. Especially if the plateau effect would appear close to the starting clause it might be an important issue for our top-down stochastic search. In the section 6.3.3.2 we suggest using a new approach to solving the issue – we use a slight modification of the evaluation function that makes the function to give better evaluation to more complex clauses. We perform this by adjusting the fitness function so that it includes a factor of small weight that promotes clauses with higher complexity.

To avoid the risk of potential overfitting this might cause we combine this with another technique – simplification of the clause prior to adding it to the model to obtain its maximal generalization that has the same coverage According to the *Occam's razor principle*, generalizing clauses before adding them to the final model should limit overfitting and should lead to better results on test data. This step is used independently of the complexity factor in the evaluation function and its general usefulness is analyzed and proved in the Section 7.3.4.

Generally, we stand here between two dangers: either we may create an over-fitted model because of preferring longer clauses that are too specific and use too many literals or we may not discover the target concept because of preferring simple clauses and being trapped in a plateau. *Our question to answer is whether preferring clause complexity during the search improves the quality of the final solution.* To answer the question we will compare models that used the complexity factor during learning with models that do not and analyze the accuracy of the final solution. We will also compare our solution to other standard plateau tackling approach inspired by the *Walk-SAT* algorithm.

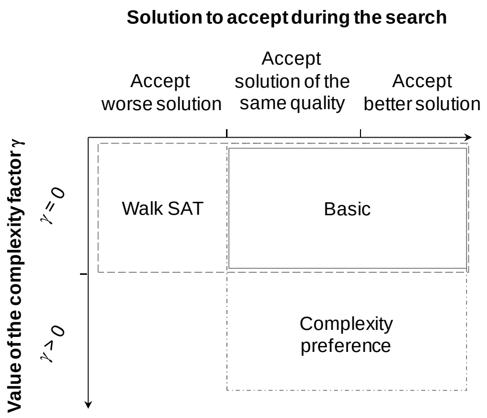
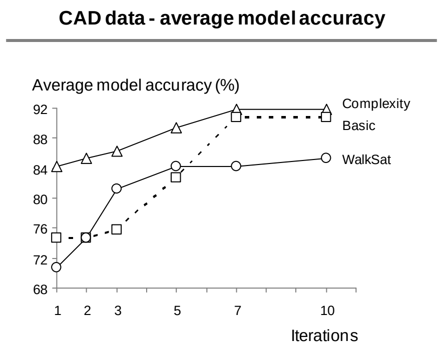

Solution to accept during the search
Accept worse solution
Accept solution of the same quality
Accept better solution
Value of the complexity factor γ
γ = 0
γ > 0
Walk SAT
Basic
Complexity preference

Figure 7.15 – Comparison of the three plateau facing techniques used in this section

In this experiment we will compare our novel approach to two other techniques (schematically compared also in Figure 7.15):

1) *Basic POEMS approach* – relies only on the basic features of POEMS algorithm described above (in other words, this is the basic approach of the POEMS system with complexity factor set to zero).
2) *Walk-SAT inspired approach* – uses the extended GSAT strategy of *plateau moves* that allows the searcher to temporarily (and with small probability) accept solution that is even worse than

the best discovered one. The probability we used is proportional to the difference between solutions with the maximum difference limit set to 20% (i.e. when a solution is worse by 10% than the best one, there is 10% probability that it still would be accepted; solutions with value of the evaluation function worse by more than 20% of the best-so-far solution are rejected automatically).



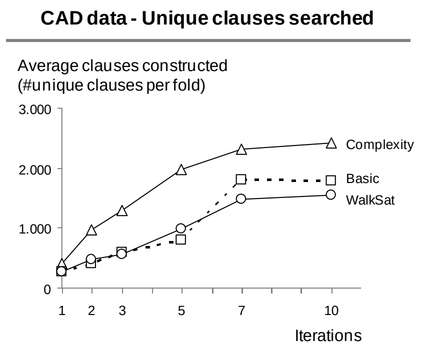
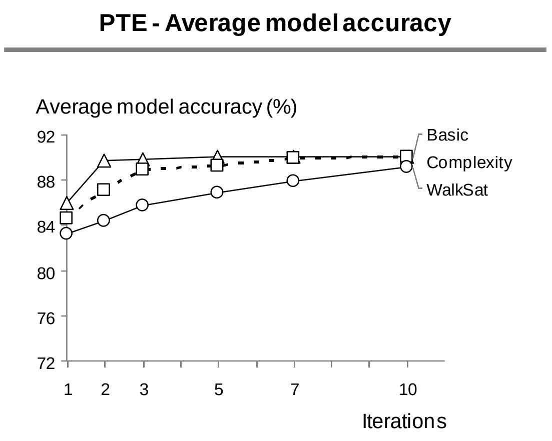


Figure 7.16 – Average model accuracy on *CAD* test data and number of unique clauses searched with the three different plateau tackling approaches

To analyze the effect of these approaches we start with the *CAD* dataset and run five-fold cross validations with each one. To get a broader picture and consider also the influence of iterations we consecutively increased the maximal limit on the iterations in the sequence 1-2-3-5-7-10 iterations and run five times the experiment for each iteration limit. The average model accuracy and average number of unique clauses generated with the three approaches are summed up in the Figure 7.16.
The chart shows that when the complexity factor is used the algorithm on average achieved higher accuracy of the final model already for lower number of iterations (almost no change after second iteration). The basic version of the evaluation function reached the quality of the complexity preferring approach for the number of 7 and more iterations, for less iterations the results are worse by 5-10pp. The results of the *WalkSAT* variant were slightly better than *Basic* version for 3 iterations but the accuracy for more iterations does not reach the levels of the other two approaches.
Comparing the number of clauses searched we can see that the complexity preference in case of this dataset leads to increased number of clauses searched. The reason is that longer clauses offer more options for further adaptations (e.g. more places to add a constant). The number of clauses generated with the *Basic* version is similar to the *WalkSat* approach.



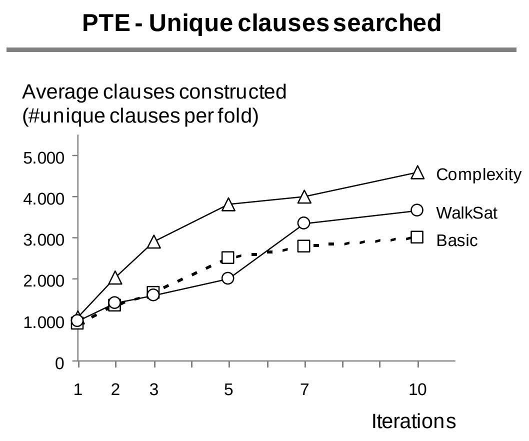
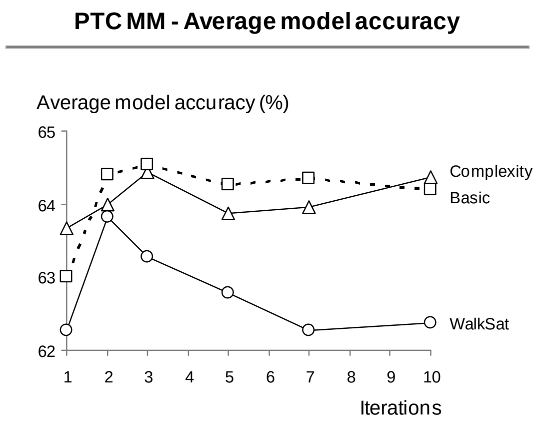


Figure 7.17 – Average model accuracy and number of unique clauses searched (*PTE* data)

We performed the same experiment also with the *PTE* and *PTC* datasets. The results are depicted in Figure 7.17 and Figure 7.18. The effect of the complexity factor on the number of clauses generated is similar to *CAD* – the number of searched clauses is higher when the factor is used. However, there appears to be no positive effect on the accuracy of the model when this factor is used. This seems to be caused by the fact that both *PTC* and *PTE* datasets do not require so long clauses for proper classification (like the *CAD* data). As shown in the Table 7.10, the final models for both chemical datasets contain clauses of much smaller complexity than are the clauses of an average *CAD* model. Therefore, the complexity factor plays more important role for the *CAD* dataset.



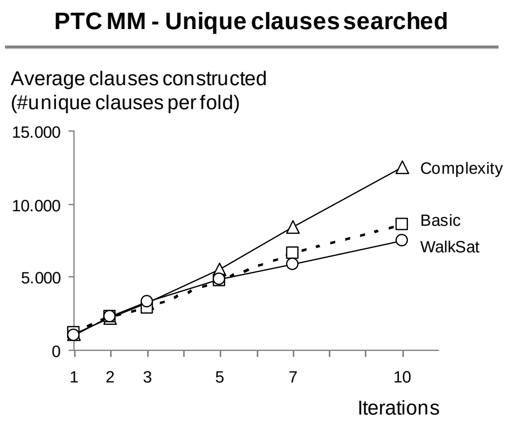


Figure 7.18–Average model accuracy and number of unique clauses searched (*PTC MM* data)

Although we cannot see any improvement in the accuracy when using the complexity factor, it has also no deteriorating effect on the model quality. Both charts show that the predictive accuracies of the final model with and without the complexity factor (the *Basic* version) are comparable. Therefore we suggest the complexity factor (combined with clause generalization – see Section 7.3.4) can be used even for new problems where the user has no information about potential plateau issues. On the contrary, the *WalkSAT* inspired approach produced the worst results for all three datasets and we think this technique is not a proper extension of the POEMS system.

Table 7.10 – Average number of predicates and constants per clause in model (without using complexity factor)

Dataset	Predicates + constants (avg.)
PTC MM	3.85
PTE	3.11
CAD	5.75

7.3.2.3 Evaluation Function - Conclusions

In this subsection we solved two questions regarding the evaluation function that shall be used in our POEMS system. We focused on selection of proper evaluation function that will be used as the main criterion during the sequence optimization and we analyzed our novel approach to face the plateau problem by adjusting the evaluation function with complexity preferring factor.

Selection of the evaluation function. Regarding the selection of the evaluation function we take an approach that is similar to other ILP systems - we search for a different evaluation function than the clause accuracy (although the main criterion against which we finally evaluate success of our modeling process is the predictive accuracy of the model over the test data). This hypothesis was justified by our experiments as the results showed that accuracy actually is not the most suitable for the search process - building the model by set-covering approach. We tested six evaluation functions taken from the literature and for each function we evaluated the final accuracy of the model on the test data by taking 5 relational datasets and running five times the five-fold cross-validation experiments. The experiments showed us that the ILP tasks we are dealing with generally fall into two categories:

1) problems in which exist patterns that non-symmetrically cover large parts of the example sets (either E^+ or E^-) and can be used for successful classification (like the *CAD* and *Trains* problem) – for these problems the optimal model contains only a few clauses and the experiments showed that for these tasks the *novelty* evaluation function leads to most accurate models;
2) problems that contain more noise and for which no patterns with such large coverage do not exist – for these problems more granular models that contain more clauses are needed and the experiments showed that e*ntropy* function gives the best results for this type of ILP tasks.

Conclusion from the experiments (see Table 7.11) is that for most of the tasks we will use the *entropy* function with POEMS as it on average gives better results than other functions. We also suggest that *entropy* is a good starting point when analyzing potential new dataset. However, for the less noisy tasks for which we know they offer patterns (clauses) covering larger numbers of the examples using we will use the *novelty* function. This means, we will use entropy evaluation function in our work except for the *CAD* and *Trains* dataset

Table 7.11 – Evaluation function to be used for each dataset

Dataset	Evaluation function
Trains	Novelty
PTC MM	Entropy
PTE	Entropy
Alzheimer memory	Entropy
CAD	Novelty
other (general setting)	Entropy

Facing the plateau issue. We compared the basic approach (without the complexity factor) to our suggestion and to an approach inspired by other plateau facing technique – the *WalkSAT* algorithm. Based on the results we will use positive complexity factor as an enhancement to our evaluation function in the case of *CAD* dataset. The experiments showed that using this factor increases the quality of the model in cases, when the model needs to be built of more complex clauses (like the *CAD* model). For other datasets, the extension neither improved nor worsened the search results. This is due to the fact that the other datasets did not require constructing clauses of higher complexity like the *CAD* dataset.

7.3.3 Search Strategy

In this section we focus on proper setting of the search strategy of our system. The main issues are what is the optimal length of the action sequence and what is the optimal number of iterations the system should operate with.

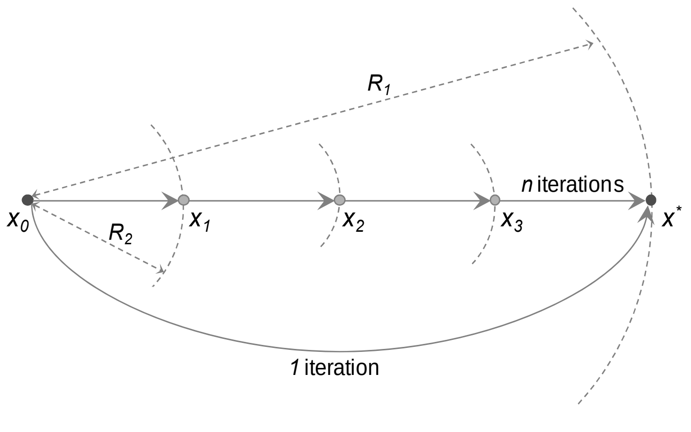


Figure 7.19 – Decision on the search strategy: optimize long sequence in 1 iteration or use short sequences and more iterations (x_0 .. start phenotype, x^* .. optimal solution)

The POEMS system is designed to iteratively construct the refinement path to the final solution represented as *meta-operator* that changes the starting clause to clause with good coverage for classification. The basic question is whether the length of the optimized action sequence should be set so that it is possible to construct this *meta-operator* in one step or it should be set with the assumption that several iterations are needed to reach the optimal solution.

Setting the proper length is therefore not a trivial task. Combined with the possibility to iterate the search we thus in front of decision between optimizing the sequence as a whole in one iteration or assembling the sequence in a step-by-step manner using a cascade of iterations. The decision is depicted in Figure 7.19. It is clear that optimizing the sequence in a single iteration mode should help skipping local optima, however, the space searched (depicted by the radius R_1) is much larger and the time needed for reaching the optimum is longer. Also, the risk of missing the correct solution is higher as well. On the contrary, using shorter sequences to search in a smaller radius (R_2 in the figure) and iterating the search on might be faster but exposes the algorithm to getting stuck in the local optima. We can see two basic risks if the sequence length is not set optimally:

- if the sequence is too short, it is possible that the algorithm will get trapped in local optimum and will not be able to reach the optimal clauses;
- if the sequence is too long, the learning process might take too much time and could lead to overfitting; also, if the search operators used do not allow for sufficiently granular search steps, the optimal solution might be "*jumped over*".

7.3.3.1 *The Optimal Sequence Length*

As we do not know the position of the optimal solution in the search space, the optimal length of an action sequence is such length that allows the algorithm to create maximum number of unique clauses useful for the search i.e. clauses with non-zero coverage. Growing the sequence size over this optimum is unnecessary as it brings only clauses that are too long, have zero coverage and as such can be hardly used for our EA-based search (all zero-covering clauses carry the lowest fitness values which makes them improper for the EA selection mechanism). *Therefore, we look for such length that maximizes the search potential while keeps the search space at reasonable size and minimizes creation of zero covering clauses.*

We will first look at the search potential of our sequences by analyzing the number of unique clauses with non-zero coverage that can be generated by sequence of fixed length. *Unique clause* means a clause that has not been generated so far by a sequence of the given length.

In the first experiment we generated 1,000 random sequences for each of lengths between 2 and 20 actions. We created corresponding clauses by applying these actions to the empty starting prototype clause. For each sequence length we check the share of the number of unique clauses generated by random generator and look for the shortest sequence length at which most of unique non-zero covering clauses are generated. We believe that the answer to the question of setting the sequence length optimally will be different for each dataset. This is because the optimum should depend mainly on the structure of the analyzed dataset in combination with the selection of the starting clause and the set of the refinement actions used (the set of actions we use in this experiment is given in Table 7.5).

Results of this experiment on the *Trains* dataset are in Figure 7.20. According to the figure, the number of randomly generated different unique clauses grows linearly with the sequence length. It is for the action length of more than 12 actions for which we can observe a small decrease to the slope of the curve. Additionally, almost all of the generated clauses have non-zero coverage (cover at least 1 example). On the other hand, the share of unique clauses is rather low when compared to other datasets (see below). For the sequence length of 6 actions the share of unique clauses generated is only 25% meaning that 100 such random sequences produce only 25 different clauses (for PTE this share is around 40%). This in the end prolongs the search time.

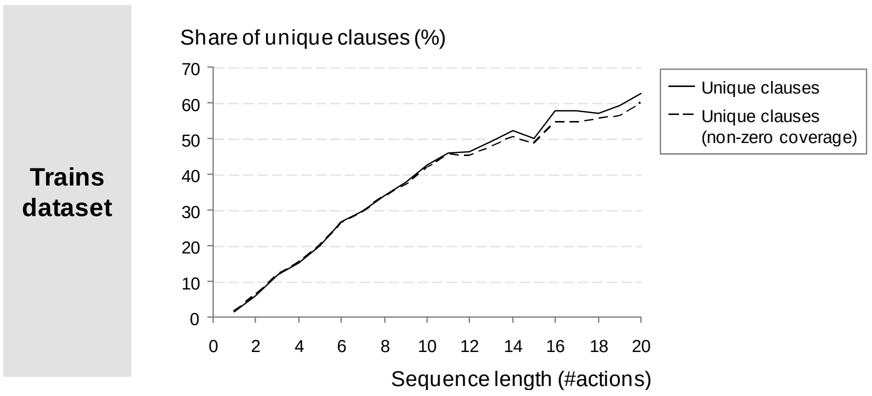


Figure 7.20 – Number of unique clauses generated by random action sequences of different lengths (unique clauses and clauses with non-zero coverage, 1,000 clauses for each length)

Based on our knowledge about the dataset, at least 7 actions are needed to reach optimally classifying clause for the *Trains* problem. Considering the linear growth of unique clauses the optimal size of the sequence should be 7-8 actions and the result should be found in single iteration but this needs to be tested by the iterated algorithm run (see the next experiment).

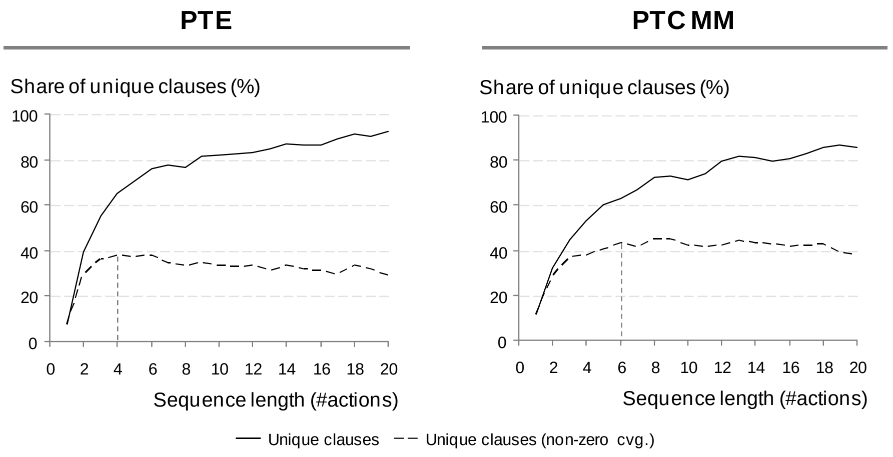


Figure 7.21 – Number of unique clauses generated by random action sequences of different lengths (unique clauses and clauses with non-zero coverage, 1,000 clauses for each length)

For the *PTE* and *PTE MM* datasets (Figure 7.21) we can observe that the share of clauses with non-zero coverage peaks at approximately 40% for the sequence lengths of 4 actions (*PTE*) respectively 6 actions (for the *PTC MM* dataset). Actions longer than this still produce more unique clauses but the growth is made through clauses with zero coverage and the share of unique non-zero covering clauses on the total number of clauses generated for each sequence length (=10,000 clauses) remains steady or even slightly decreases. Therefore, we expect the optimal sequence length for these datasets to be up to 4 actions for the *PTE* dataset and up to 6 actions for the *PTC MM* dataset.

Similar situation like for the *PTC MM* dataset is also for the *Alzheimer memory* dataset (see right hand side of Figure 7.22) where the number of unique clauses with non-zero coverage peaks at the length of 6 actions. The peak level is even higher - 65% of clauses generated with sequences of 6 actions are unique and have non-zero coverages. The share of non-zero covering clauses for sequences longer than 6 decreases meaning that with sequence length growing over 6 the probability of generating useful clause at random decreases as well (for 16 actions it is only 40%).

For the *CAD* data (see Figure 7.22, left hand side) we can see behavior similar to that of the *Trains* dataset. Compared with other datasets, the share of unique clauses is again rather low (only 15% at the sequence length of 6 actions) which is caused by the fact, that most of the examples share similar structure and often differ only in few literals or in the values of the constants used.

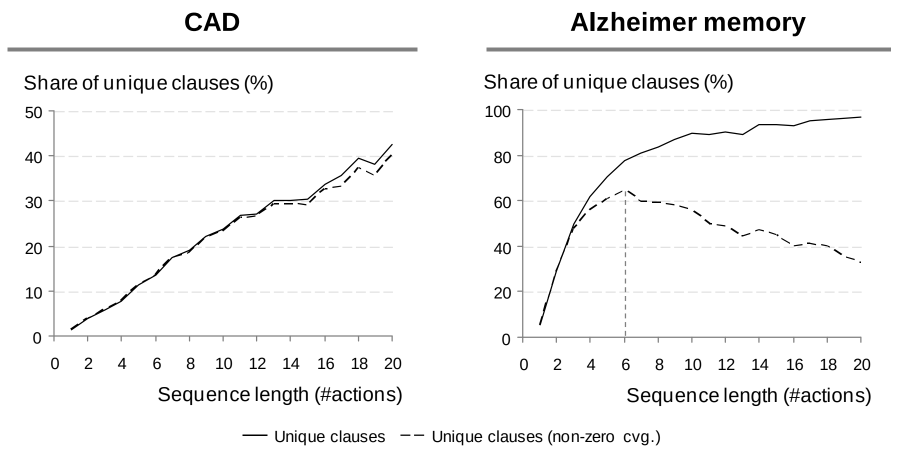


Figure 7.22 – Number of unique clauses generated by random action sequences of different lengths (unique clauses and clauses with non-zero coverage, 1,000 clauses for each length)

7.3.3.2 *Sequence Length in the Complete Algorithm Run*

In this section we briefly analyze the influence of the sequence size in combination with different number of algorithm search iterations. The optimal setup is the one reaching model with the best predictive accuracy but also searching the smallest number of clauses.

The selection is actually a complex problem. There are two factors that influence selection of the optimal sequence length. The first one is the iterative mode in which the POEMS system runs. In each iteration, it creates action sub-sequence that prolongs the refinement sequence that modifies the starting phenotype pushing the search to start from different position. The effect of these iterations might change the optimal sequence length the indications of which were given in the previous section. Additionally, the number of iterations is not a hard limit – the search is immediately stopped once a clause with is discovered that covers only one type of examples (either only positive or only negative examples).

The second external effect is caused by the fact that the system uses the set-covering approach - discovering a clause in one step of set covering means the examples covered by the discovered pattern are removed from the example set and search continues on an altered landscape defined by the evaluation function). This also means that the optimal sequence length in the next set-covering steps might actually be different from the one selected for the first step.

Considering these factors we suggest using *variable sequence length* in our system instead of fixed length. We try to define the lower and upper boundary of the sequence size within which the optimization algorithm (GA in our case) can select the most proper one. Our hypothesis is that *using the variable sequence length offers flexibility that is required to reach better results as it should enable the genetic algorithm to accommodate the solution to the actual problem complexity and latest form of the actual prototype solution*.

In the experiment in this section we will analyze how the number of iterations along with the limit on the sequence size influences the final predictive accuracy of the model. We will test both fixed sequence and variable sequence length. The length limits of the sequences are based on the results of the previous subsection (showing the optimal search capabilities in the range of 4 to 6 actions):

- *Variable length.* We test sequences of following lengths 2 to 4 actions, 2 to 6 actions, 2 to 10 actions and 5 to 10 actions. For the *Trains* dataset we use 7 actions instead of 6 as we know the optimally classifying clause has the length of 7 predicates. The motivation for this selection of the sequence lengths is that it allows to use the optimal length as well as to test both shorter clauses (sequences with minimal length of 2 actions) and longer clauses (sequences with maximal limit set to 10 actions).
- *Fixed length.* We test the fixed length of 6 actions (7 for *Trains*) to see the comparison of the results of fixed length sequence to the variable lengths.

The results of the first experiment with the *Trains* dataset are shown in Figure 7.23. The figure shows that best results were recorded for the sequence length of 5 – 10 actions for which the optimal pattern was always discovered already after the first iteration. Also the sequence length of 2-10 actions shows good results, especially when using more iterations than one. The results when using fixed chromosome size (7 actions) were satisfactory only for the search with one iteration. Apparently, when the search is performed in more iterations, the algorithm has difficulties to accommodate the sequence so that it could change the working clause only in a small extent. Utilization of a short sequence (2-4 actions) also did not bring satisfactory results which shows that multiple iterations cannot always compensate for too short sequence length.

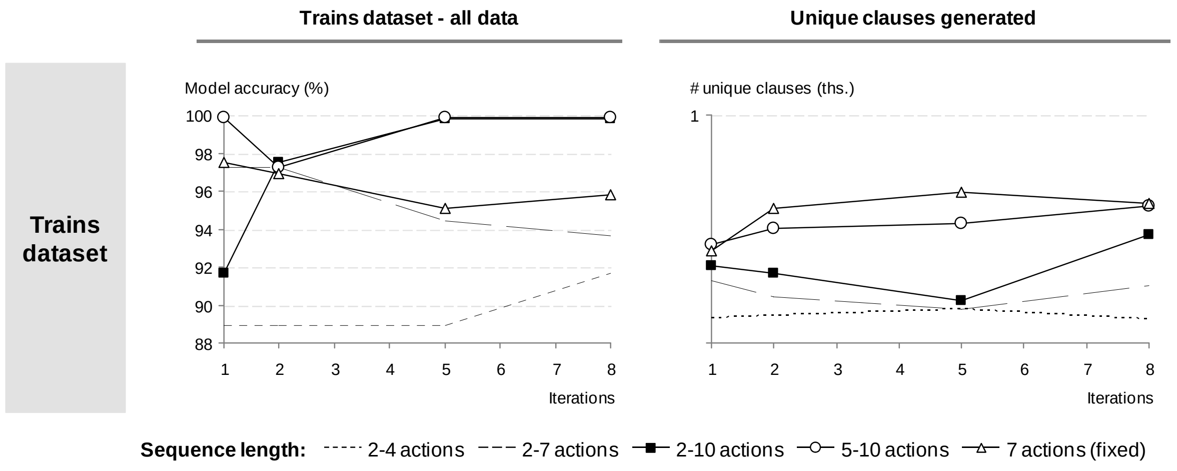


Figure 7.23 - Effect of different sequence lengths on model quality and clauses searched, Trains dataset (all data, averages from 25 runs)

When we look at the number of clauses searched we can see, that it remains more or less unchanged regardless of the number of iterations. This indicates that in most of the cases the algorithm discovered clauses with one-sided coverage much earlier than the maximum iterations limit was reached.

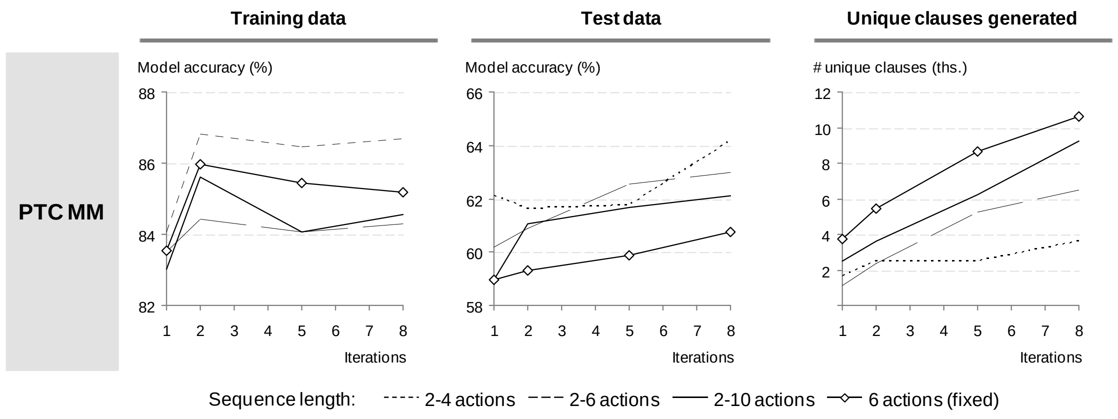


Figure 7.24 – Accuracy of the model for different sequence lengths and number of iterations (showing averages per 1 fold of five-fold cross-validation)

The results of the experiment with the *PTC MM* dataset are summarized on the charts below in Figure 7.24. The charts show that the most accurate model on the test data is produced by the shortest action sequence (2-4 actions) in combination with 8 iterations. This supports the knowledge we have about this problem that this dataset is best classified with short clauses. We can also see that the fixed sequence shows the worst results although it searches the most clauses. Based on this experiment we will use variable sequence length with the upper length limit of the sequence 4 actions.

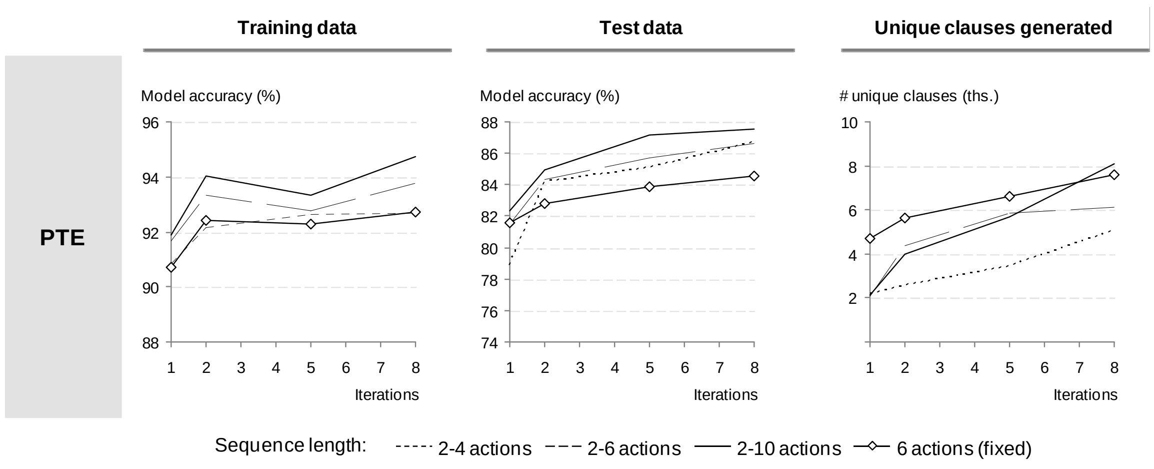


Figure 7.25 – Accuracy of the model for different sequence lengths and number of iterations (showing averages per 1 fold of five-fold cross-validation)

The results for *PTE* dataset are in Figure 7.25. The results of the different sequence lengths are comparable to the best accuracy produced in this case by the 2-10 actions long sequences. Similarly to other datasets, the shortest action sequence also produces the least number of clauses searched. Therefore it can be the second option for this dataset in case when we wanted to achieve short time of the learning process. Also here the predictive accuracies of models constructed with the fixed length are worse than the results of models produced by variable sequences.
Considering the number of unique clauses searched (where the short sequence short the lowest amount of clauses) for this dataset we will use the lengths of 2-6 or 2-10 actions in combination with 5 or more iterations.

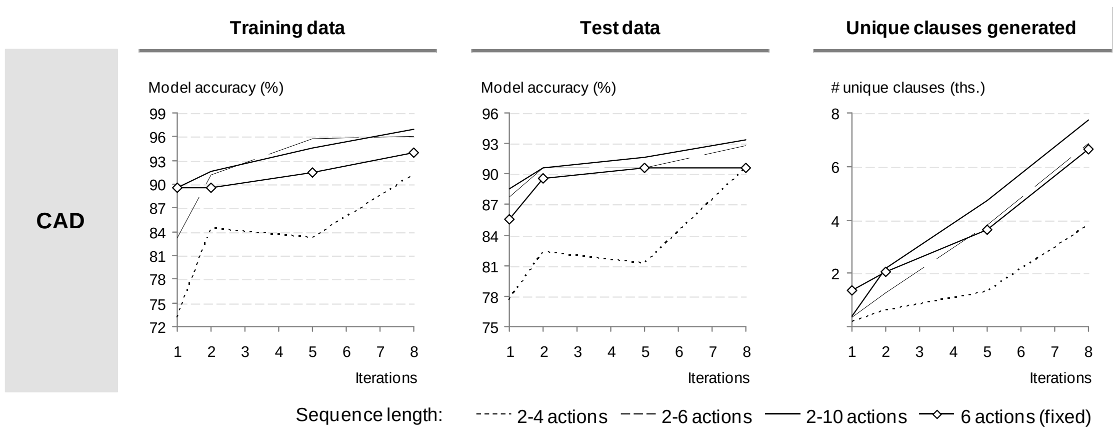


Figure 7.26 – Accuracy of the model for different sequence lengths and number of iterations (*CAD* data, showing averages per 1 fold of five-fold cross-validation)

For the *CAD* dataset (see Figure 7.26) we can see that the best results are reached with the sequence of 2 to 10 actions. This is in line with our knowledge about the dataset that longer clauses are needed in

the model for good example classification. Also here the models created with shorter sequences with up to 4 actions show worst performance although the quality improves only after using 5 iterations and more. The quality of models produced with fixed chromosome is comparable to the models created with sequence lengths 2-6 and 2-10 actions.
Considering the number of unique clauses searched (where the short sequence searched the lowest amount of clauses) for this dataset we will use the lengths of 2-6 or 2-10 actions in combination with 5 or more iterations.
The experiment on the *Alzheimer memory* dataset is summarized on the charts in Figure 7.27. While the fixed sequence length gives in this case the worst results (both for training and test data), other three chromosome sizes are comparable with the 2-10 length being slightly better than other two sequence lengths. Both for the training data and the test data the accuracy of the model slightly increases with the number of iterations reaching levels around 93% for models built with search 8 iterations. Considering the number of unique clauses searched (where the short sequence searched again the lowest amount of clauses) the lengths of 2-4 or 2-6 actions in combination with 5+ iterations seems to be good option for the *Alzheimer* dataset.

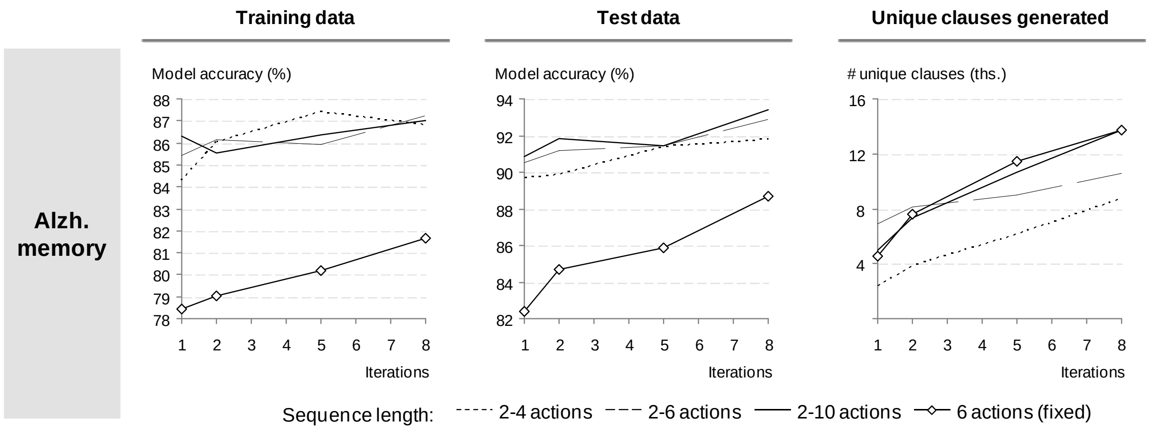


Figure 7.27 – Accuracy of the model for different sequence lengths and number of iterations (*Alzheimer memory* dataset, showing averages per 1 fold of five-fold cross-validation)

7.3.3.3 *Search Strategy – Conclusions*

In this subsection we briefly analyzed the optimal set-up of the search parameters – action sequence length and minimal number of search iterations needed for each dataset. The conclusions of the analyses based primarily on the predictive accuracy of the final model on test data are summarized in Table 7.12 (these are also the values we will use for these datasets in this work). We suggest using the variable sequence length as this gives the searcher more flexibility and in the experiments resulted in models with higher accuracy. The minimal sequence length limit we use is length of 2 actions (except the *Trains* dataset), the upper limit depends on the analyzed task.

Values of the sequence lengths and limit on the number of iterations that will be used for each dataset are summarized in Table 7.12 below.

Table 7.12 – Identified search strategy for the datasets

Dataset	Sequence size	Max. iterations
Trains	5-10 actions	1 iteration
PTC MM	2-4 actions	8 iterations
PTE	2-10 actions	5 iterations
Alzheimer memory	2-6 actions	5 iterations
CAD	2-10 actions	5 iterations
other (general setting)	2-10 actions	5 iterations

7.3.4 Clause Generalization Step

In this subsection we will briefly focus on proving the usefulness of clause generalization that we use prior to including the clause into the predictive model.
From our experiments with the search strategy it came out that the POEMS algorithm produces better results when 2 or more iterations are used. The experiments also showed that increasing number of iterations leads to increase in accuracy over the training example sets which might bring the issue of overfitting. In this section we will briefly focus on our strategy to tackle this risk that might happen in cases when the resulting action sequence is "too long" and leads to unnecessarily large clauses entering into the model.
To face overfitting we include the generalization step before adding the clause to the model. In this step we take the clause corresponding to the best discovered clause modifying sequence and start greedily removing predicates, constants and variables until the value of the evaluation function for the clause measured on the training data changes negatively (e.g. decreases when maximizing or increases when minimizing). This generalization step should bring two effects:

1) limit overfitting – work against construction of long meta-sequences that lead to construction of too specific clauses (especially when preferring more complex clauses during the search to avoid the plateau effect), and
2) improve the clause quality – helps e.g. in cases when the refinement sequence length is too long jumps over the optimal solution (or cannot produce solution that is simple enough).

To confirm this expectation we run the five-fold cross validation experiment to analyze the accuracy of the final model with and without the generalization step. The effect of generalization before adding the clause to the model is notable on comparison shown in the Figure 7.28 (5 times five-fold cross-validation). From the figure it is clear that the average accuracy of the models on test data is higher compared to situation when complexity coefficient is used but the generalization step is excluded.

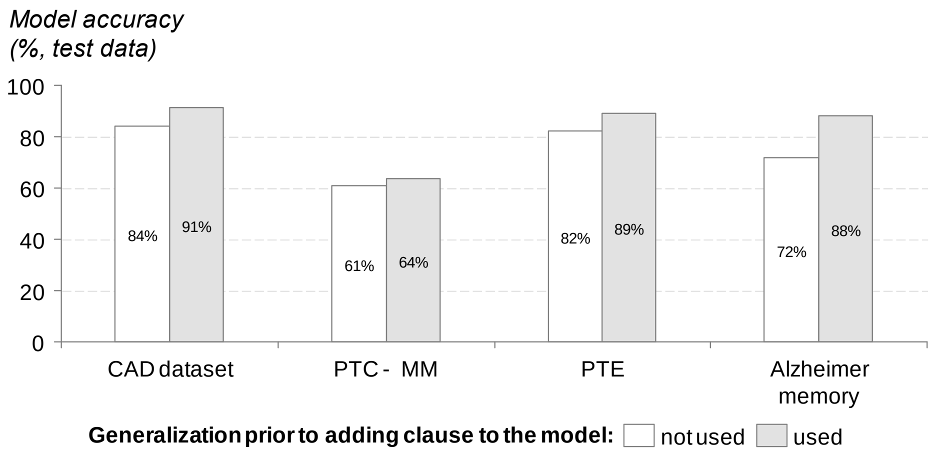


Figure 7.28 – Average model accuracy on test data with and without generalization of clauses prior to adding them to the model

7.3.5 Layer 2: Conclusion

In this section we analyzed the influence and optimal set-up of the main search parameters on the outcome of our POEMS algorithm for ILP tasks. Namely these were: the sequence optimization algorithm, the evaluation function (selection of the function and the complexity preference factor), the search strategy (sequence length and number of iterations) and generalization step prior to adding clause to the model.

Summary of the conclusions is given in Table 7.13. Based on the results of our experiments we decided to use genetic algorithm as the sequence optimization algorithm in POEMS. We will use two evaluation functions – novelty for the problems that contain less noise and can be optimally solved with models of smaller sizes (*Trains* and *CAD* datasets) and entropy for the remaining ILP problems. We will use the variable sequence length for all the problems at hand as this showed better results in most of the cases. The choice of the limit of the number of iterations and the length of the action sequence depends on the dataset. Generally, we will use at least 5 iterations and upper limit on the sequence length in the interval between 4 to 10 actions. We will also use the generalization of the discovered clauses prior to adding them to the model as this step proved to increase the accuracy of the final model.

Table 7.13 – Summary of issues and proposed solutions regarding sequence optimization

Issue	Question	Solution	Reason
Optimization Algorithm	Select the most efficient optimizer	GA	Predictive accuracy, size of the space searched (efficiency criterion)
Evaluation function	Select the proper evaluation function to guide the search	Entropy and novelty (depends on the dataset)	Predictive accuracy, size of the space searched (efficiency criterion)
Search strategy	Find the optimal balance between the number of iterations and size of the individual action sequences	Use variable sequence size, the upper length limit depends on the dataset	Predictive accuracy, size of the space searched (efficiency criterion)
Overfitting	Test importance of generalization prior to adding clause to the model	Generalization will be used	Predictive accuracy

7.4 The Model Construction Parameters

In this section we focus on *Layer 3* of our algorithm and on the parameters that influence the greedy set-covering approach that is used to construct the final model. The main parameters used in our implementation that influence the efficiency of the theory construction layer are: model size (number of clauses) and minimal clause coverage (see Table 7.14 for summary and the criteria measured in each experiment):

– *Minimal clause coverage.* Sometimes the POEMS algorithm produces clauses that show good classification power (e.g. cover examples of one class only) but have extremely small coverage. Combined with the set-covering approach we use in the system there is high risk that the classification quality of these clauses on the test set (or on other new examples) will be low. This can be avoided by using the minimal clause coverage constraint. We expect that utilization of this threshold should bring two effects – it should limit the risk of overfitting and also lead to finishing the search in shorter time (as the algorithm should produce clauses with higher coverage and fewer clauses should be needed to cover all the training examples).

  The main issue that is analyzed also here is to detect the borders within which the final classification model gets neither too detailed nor too general. Running the system with different thresholds on minimal clause coverage should answer the question.

– *Model size.* The final model comprises of a set of classifying hypotheses in the form $Class_C \leftarrow C$, where C is the clause discovered on *Layer 2* and $Class_C$ is the class of the majority of examples covered by C. The first question is, whether the learning time can be decreased by limiting the model size while keeping model of the same classification quality. The second question is whether putting a constraint to cap the maximum model size can help to avoid overfitting as the model will not get too detailed. To learn this we will run the algorithm several times with different limit on the final model size and compare both the final predictive accuracy of the model and the size of the space searched during the model construction phase.

Table 7.14 – Summary of issues of the model construction layer (*Layer 3*)

Issue	Question	Options	Evaluation criteria
Minimal clause coverage	Find proper threshold value to limit overfitting and speed up the search	Different values of minimal clause coverage limit (minimal share of examples the clause must cover)	Accuracy of the final model, average model size, total number of generated unique clauses (~time needed)
Model size	Find proper model size to limit overfitting and speed up the search	Different limits of maximal model size (max. number of clauses in the model)	Accuracy of the final model, total number of unique generated clauses (~time needed)

To assess the effects of these constraints we designed following experiment for both of them. For each of the datasets we run several batches of five times five-fold cross validation. In each batch we decrease the limit of the constraint followed and record the accuracy of the final model (both training and test data), its size and the number of unique clauses constructed.

7.4.1 The Minimal Clause Coverage

In this subsection we focus on setting the minimal coverage constraint. We expect there are two main effect of this constraint – avoiding model overfitting and pruning the search space so that the search will be faster. The risk of overfitting is high especially when using the entropy function in combination with noisy real world datasets. In this case our learner that uses the greedy set-covering approach tends to produce models that consist of a large amount of extremely specialized clauses each covering only a few examples (only one example in the extreme case). Such models are over-fitted with low generalization ability. Using the minimal coverage limit should work against this risk.
The second effect of this constraint is pruning the search space and thus shortening the learning time. When the constraint is used in the sequence optimization phase (and not at the end of it for pure filtering), the searcher can prune large parts of the search space that contain clauses the coverage of which is too low. On the contrary, the higher the limit is set the fewer clauses can fulfill the constraint and missing the proper granularity might mean missing the optimal model.
To set the proper value we subsequently changed the minimal coverage limit in the sequence 50% - 20% - 10% - 5% - 2% - 1.5% - 1% - 0% of training examples covered. We did not include the *Trains*

dataset into the analysis as there is only one perfectly classifying concept for this problem and thus the minimal coverage limit has no influence on the model quality. It should be also noted that because we use the minimal coverage threshold directly in the sequence optimization algorithm (GA) we can avoid producing these clauses already on the algorithm search level (the algorithm disqualifies the sequences that lead to such clauses already when such sequence is constructed during optimization).

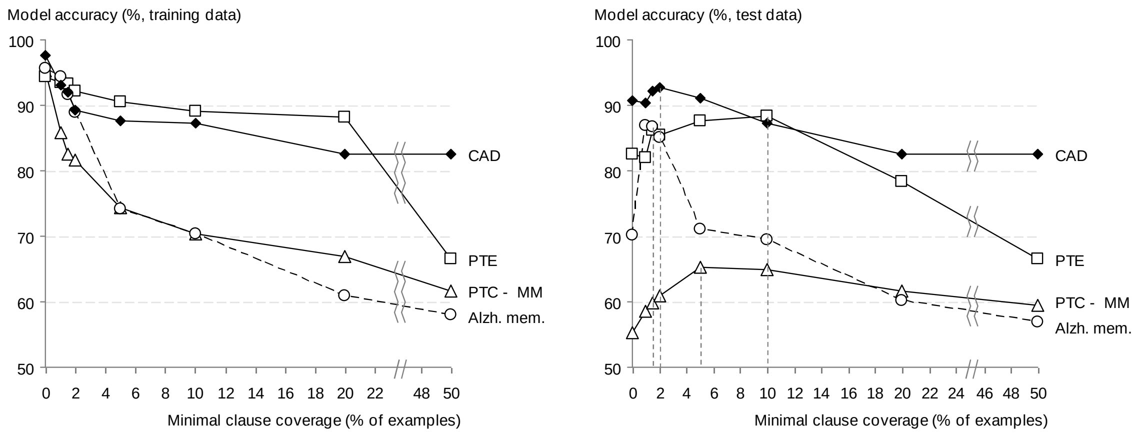


Figure 7.29 – Effect of minimal clause coverage on model accuracy (training and test data averages, all datasets)

The effect of the constraint on the accuracy of the model for the all datasets is shown in Figure 7.29. While the accuracy of all models on the training data rises steadily with the decrease of the constraint for all the datasets, the accuracy on the test data peaks in the range of 2% - 10% covered examples. Thereby, the chart shows nicely the effect of overfitting for low values of the minimal coverage constraint – the decrease of predictive accuracy for the values below 2% is significant. The only exception is the CAD dataset for which there is only a small decrease on the minimal coverage level of 0%. This is due to the fitness function that is used for this dataset – novelty. Unlike the entropy function, the novelty function leads the searcher to prefer clauses with larger coverage and therefore the risk of overfitting is lower when this function is used. For all datasets we can see that if the minimal coverage constraint is set to 0%, the performance of the model on the training data is the best. On the other hand, the comparison on the test data shows that the accuracy of these models that are highly accurate on the training data is actually lower than accuracy of models built with higher value of the coverage constraint. This proves the fact that the model is over-fitted and its generalization ability is not sufficient.

When we look on the other side of the chart axes we can see that when the limit is set too high (20% and higher), the model accuracy is lower than the accuracy for the limit set in region around 5% of the training examples. This is a sign that the coverage of clauses is too large and therefore not sufficient to produce model with sufficient granularity. The experiment shows that the optimal level of the threshold lies between 2% and 10% depending on the dataset.

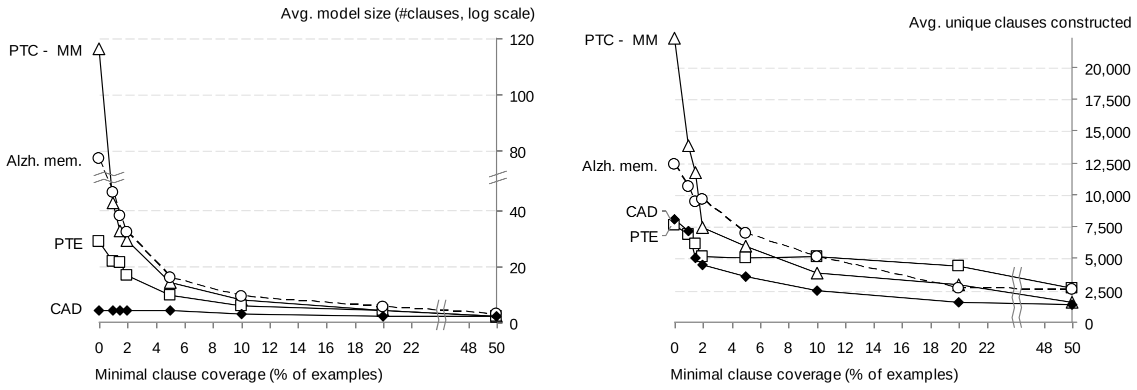


Figure 7.30 – Effect of minimal clause coverage on the number of constructed clauses and final model size (showing average numbers per one fold in cross-validation)

Before making the decision on the value of the constraint we briefly analyzed also the complexity of search each threshold value brings and examined the number of unique clauses created under each constraint (see Figure 7.30). On the chart we can see that the number of unique clauses starts to rise enormously for the values below 2% (especially for the *PTC* dataset). It seems that allowing the search to look for clauses with less than 2% coverage hinders the algorithm to both produce a "*good enough*" model and keep the number of explored clauses at a reasonable level.
Based on the results of the experiment we will set the minimal coverage constraint to the level of 10% of training examples for the *PTE* dataset, 5% for the *PTC* datasets and 2% for the *Alzheimer* datasets (see Table 7.15).

Table 7.15 – Summary of issues and proposed solutions of the model construction layer

Dataset	Trains	PTC MM	PTE	Alzheimer mem.	CAD
Minimal clause coverage	0%	5%	10%	2%	1%

7.4.2 Limiting the Maximal Model Size

In this section we focus on setting the maximum model size limit. This means finding a proper balance between the accuracy of the model, learning time and potential overfitting. *Our hypothesis is that the constraint should not be set too high to allow for sufficient model granularity but also not too low to avoid extreme runtimes and overfitting.*
The model is obtained by repeatedly running the POEMS algorithm and excluding the examples covered by last discovered clause from the set of examples that have been not covered so far. Because the algorithm stops automatically if the training example set is empty it is not possible to force the model to contain some externally given exact number of clauses. However, we can limit the maximum number of clauses the model can contain. If the learner reaches the limit it exits the search.

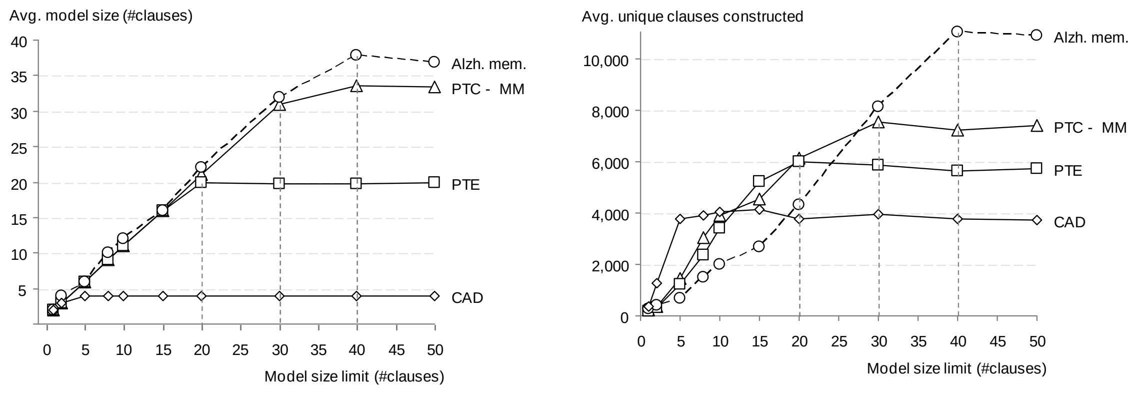


Figure 7.31 – Average final model sizes and number of clauses searched for different values of max. model size constraint (showing avg. values per fold)

In the experiment we run five times five-fold cross validation and for each batch we sequentially change the maximum limit of clauses the model can contain in the sequence 1 – 2 – 5 – 8 – 10 – 15 – 20 – 30 – 40. It shall be noted here that the model always contains one more clause above the limit - the empty clause that "*closes*" each model by classifying any remaining example into the majority class. This clause was not included in the limit). We used also the minimal clause coverage limit set to values determined in the previous chapter (see Table 7.15).

We first briefly look at the relation of the model size limit and the actual average model size for each dataset (see Figure 7.31). The last clause that is used to classify any example to the majority class is also included in the model size. The chart clearly shows that after reaching the optimal size the model size does not grow any more. The optimal size for *CAD* data is 4 clauses, for the *PTE* dataset is around 20 clauses and the optimal size for *PTC* and *Alzheimer* datasets is around 35 clauses.

The chart on right hand size of Figure 7.32 gives the information about the number of unique clauses searched at each model size limit shows that after the optimal size is reached the number of unique clauses searched does not grow any more. This means that there is strong relation between the size of the final model and number of clauses that need to be searched when constructing the model. It is clear that limiting the model size below the maximal values for each dataset cuts also almost proportionally the learning time. The question remains what would this restriction mean towards the accuracy (will be specific for each dataset) as this is our main criterion for evaluation of the learning algorithm output quality.

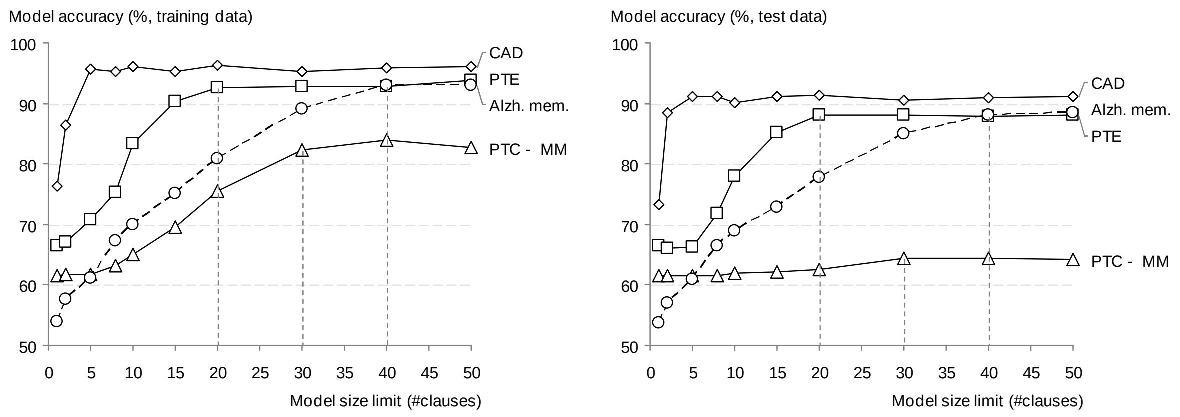


Figure 7.32 – Model accuracies as function of final model size (in number of clauses, empty clause that closes each model is not included), training and test data

Figure 7.32 shows the relation between maximal model size and model accuracy (measured both on training and test data). The progression of the accuracy on the training dataset is as expected – the closer the learner gets to the optimal model size the higher is the accuracy on the training data. The results on the test data show that also the predictive accuracy of the model on the non-seen data examples grows with the growing size of the model up to reaching the optimal size of the model. However, the charts do not show any signs of overfitting. This means that when overfitting clause is regulated by the minimal coverage constraint there is no need to use model size for the same purpose as well.

According to the experiment it seems that there is clear relationship between the model size and both the accuracy of the model and the size of space searched. Constraining the model size means reaching lower quality of the model but the learning process is terminated in shorter time. However, the space searched (expressed in unique clauses produced during learning) grows rather linearly than exponentially and even for the maximal model sizes it stays in moderate numbers. Additionally, the output does not show signs of overfitting when the model size is left unconstrained as this issue seems to be solved by setting the minimal clause coverage constraint.

The question of setting proper model size limit remains open only for the *PTC MM* data for which it is difficult to find a good classifier and the model quality increases only very slowly. However, also the state-of-the-art classifiers tend to have problems with this dataset and do not reach accuracy over 65% (see chapter 7.5 on comparing POEMS with other learners). Based on this experiment we will not use the model size constraint for any of the ILP tasks analyzed in this work.

7.4.3 Layer 3: Conclusion

In this section we analyzed the effect of two basic ILP constraints on the output of the learning algorithm – the minimal clause coverage and the minimal model size. The main question was whether to use and how to set these parameters to avoid searching excessive search space, avoid overfitting but still obtain models with high predictive accuracy. Results of this subsection are summarized in the Table 7.16 below.

Table 7.16 – Summary of issues and proposed solutions of the model construction layer

Issue	Question	Solution taken	Reason
Minimal clause coverage	Find proper threshold value to limit overfitting and speed up the search	Different values for each dataset (see Table 7.15)	Based on the predictive accuracy of the final model (with considerations of the total number of unique generated clauses)
Model size	Find proper model size to limit overfitting and speed up the search	Do not limit the model size	Based on the predictive accuracy of the final model (with considerations of the total number of unique generated clauses)

The experiments showed that utilization of the minimal coverage constraint is useful as it increases the ability of the models to generalize. Depending on the dataset, the limit should be set to 2% or more. On the other hand, the model size constraint has no effect on potential overfitting. The only question is whether to use this constraint to speed up the learning process. As the predictive accuracy is the main criterion for us and the accuracy rises until the system reaches the number of clauses optimal for the problem at hand we decided not to use this constraint in our modeling.

7.5 Comparing POEMS Accuracy to Other ILP Algorithms

In this section we the predictive accuracy of models built with our POEMS-based ILP system to the state-of-the-art ILP algorithms on the ILP problems we introduced in the beginning of this chapter (*PTE*, *PTC, Alzheimer* and *CAD* problems). Unfortunately, we were not able to obtain codes or implementations of other similar EA-based ILP systems to perform the comparison by ourselves. Therefore, when comparing our POEMS based approach with other state-of-the-art ILP solvers, we took the best results presented in the literature and compared these presented predictive accuracies with results of our systems. The advantage of this approach is that it ensures that the presented accuracies are the best possible the systems were able to achieve (tested by the authors of the systems).

Table 7.17 – Set-up of the POEMS algorithm for different datasets

Dataset	Evaluation function	Sequence size	Min. iterations	Min. clause coverage	Max. model size	Optimization algorithm
PTC MM	Entropy	2-4 actions	8 iterations	5%	Uncon-strained	GA (pop. size=50, generations=50, 1-point-crossover, p_{Xover}=0.8, p_{Mut}=0.1)
PTE	Entropy	2-10 actions	5 iterations	10%		
Alzheimer memory	Entropy	2-6 actions	5 iterations	2%		
CAD	Novelty	2-10 actions	5 iterations	-		

The parameters of the algorithm we used for building the classifier model for each dataset are summarized in Table 7.17. Because the results presented in the literature use the ten-fold cross validation we used also ten-fold cross-validation with our system.

From the tables below it can be seen that POEMS algorithm performs at least at level that is comparable to the other ILP systems. POEMS results are in all cases superior to the standard ILP systems (Aleph, Progol) and are on the same level as similar EA-systems (PTE) or kernel based systems (PTC). For the Alzheimer datasets, the results of POEMS even

PTE datasets. The Table 7.18 displays the predictive accuracies obtained on the PTE Mutagenesis dataset from comparable ILP algorithms. For this dataset we are able to compare POEMS with two other similar approaches – the EA-based system ECL and the system Foxcs that uses refinement actions (like our approach).

Two variants of this dataset were used for the comparison – except the basic one that uses all the predicates available as introduced above (including numerical data) noted as *PTE* and second one using only structural data of the compounds (predicates *atom/5* and *bond/4* only, no additional predicates) noted here as PTE_{STR}.

A 10-fold cross validation was carried out with POEMS to give an average accuracy of 83% for the PTE_{STR} data and 91% for the original *PTE* dataset. From the summary in the table we can see that this is comparable to the results of other learning systems on the same data set as gathered in (Mellor 2008). The slight loss to ECL – LSD_f algorithm seems to be due to the fact that we did not focus our approach directly to handle the numerical data and did not implement any special discretization technique. The $ECL - LSD_f$ approach that uses special discretization algorithm for pre-processing of the numeric properties prior to searching for the FOL concept.

Table 7.18 - Comparison the Algorithms, PTE dataset

Algorithm type	**Accuracy**	
	PTE_{STR}	**PTE**
Progol (Divina 2006)	0.82	0.88
POEMS-GA	**0.83**	**0.91**
ECL (Divina 2006)	-	0.88
ECL - LSDf (Divina 2006)*	-	0.93
Foxcs (Mellor 2008)	0.84	0.87

PTC datasets. Our research in the literature did not discover any EA-based ILP learner that was used to analyze the PTC datasets. The Table 7.19 refers to the predictive accuracies obtained on the four *PTC* sub-datasets and except POEMS shows the results of kernel-based ILP learner (Fröhlich et al. 2005) which is an approach that uses optimal assignment kernels to build the classifier. According to our search in the literature, predictive accuracy reported for this last approach is the highest presented for the results that use the 10-fold cross-validation to estimate the algorithm predictive accuracy. The comparison shows that the POEMS algorithm is comparable to the Kernel-based system and is better than the results of the standard Progol ILP system.

Table 7.19 - Comparison the Algorithms, PTC dataset

Algorithm type	Accuracy			
	MM	FM	MR	FR
Progol (Fröhlich et al. 2005)	0.64	0.64	0.56	0.65
Kernel (Fröhlich et al. 2005)	0.67	0.61	0.65	0.66
POEMS-GA	**0.65**	**0.62**	**0.65**	**0.68**

CAD dataset. The dataset was analyzed so far only by the standard Progol algorithm. Results comparing the POEMS run with Progol (of the 10-fold cross-validation) in Table 7.20 show that the results of POEMS-GA algorithm are better than accuracy of Progol as reported in (Žáková 2007). According to findings in (Žáková 2007), the accuracy of Progol is lower due to the fact that Progol is unable to find clauses with sufficient lengths within reasonable time. This supports the notion that implementation of POEMS in ILP is especially suitable for more complicated domains.

Table 7.20 - Comparison the Algorithms, CAD dataset

Algorithm type	Accuracy
POEMS-GA	**0.93**
Progol (Žáková 2007)	0.81

Alzheimer datasets. Also for this problem, the research in the literature did not discover any EA-based ILP learner that was used to analyze the *Alzheimer* dataset. Therefore we compare POEMS system with two latest classifiers that have been published in the literature. The latest state-of-the-art result come from the κFOIL system - the FOIL algorithm combined with kernel based approach (Landwehr 2010). This algorithm integrates FOIL with kernel methods implementing a dynamic propositionalization approach. The FOIL search is driven by the performance obtained by a support vector machine based on the resulting kernel. The second approach presented by (Huynh 2011) combines the Markov networks with Aleph (the ALEPH++ algorithm).

Comparing the results stated in these latest works (see Table 7.21, 10-fold cross validation) with POEMS we can see that for three out of the four datasets the POEMS algorithm reached the best results. It is only the *Alzheimer Acetyl* dataset for which the POEMS algorithm gives slightly worse solution than the κFOIL system.

Table 7.21 - Comparison the Algorithms, Alzheimer datasets

Algorithm type	Accuracy			
	Acetyl	Amine	Memory	Toxic
Aleph (Landwehr et al. 2010)	0.74	0.70	0.69	0.91
κFOIL (Landwehr et al. 2010)	0.91	0.90	0.80	0.94
NFOIL (Landwehr et al. 2010)	0.81	0.86	0.73	0.89
ALEPH++ (Huynh 2011)	0.82	0.87	0.73	0.90
POEMS-GA	**0.87**	**0.90**	**0.88**	**0.95**

7.5.1 Comparison with Other Systems - Conclusions

In this subsection we compared the results of our algorithm to results of other state-of-the-art ILP learners. The main criterion was the predictive accuracy measured by 10-fold cross validation. We tried to compare mainly with other EA-based learners however we were not successful in obtaining the programs form the authors. Therefore we used results of the latest state-of-the-art ILP systems as presented in the literature.

The comparison shows that our POEMS based ILP learner is a competitive approach to other ILP systems. For all of the problems the results are comparable or even better than those of other ILP learners. This validates the effectiveness of POEMS as a method for evolving rules in first-order logic with utilization of EA-based search.

Comparison of the results of POEMS with the standard ILP systems (Aleph or Progol) shows that POEMS outperforms these systems for all the datasets. Comparison to other systems that reach the best results presented in the literature shows that our system reaches the level of the state-of-the-art learners and for some datasets even exceeds these. To be more concrete, the results for the *PTE* and *PTC* datasets reached by POEMS are better than results obtained from the Progol system and comparable to the results of state-of-the-art learners, the results for the *CAD* dataset are better than the results reached by Progol system and the results for the Alzheimer datasets are better than both standard Aleph system and the latest *kFOIL* system.

The differences in the predictive accuracy rates of the systems may be due to variations in their particular inductive and language biases. For example, more sophisticated methods for handling numerical attributes were used by the *ECL* learner, which may also partially account for its better performance on the *PTE* dataset.

7.6 Conclusions

In this chapter we focused on the experimental evaluation of the various components of our POEMS-based ILP system. We used the basic structure of the algorithm that stratifies its function into three layers (see Figure 6.2) – for *Layer 3* we examined the effect of context utilization, for *Layer 2* we analyzed the basic sequence optimization parameters and for *Layer 1* we analyzed the model construction parameters. For the experiments we used five well-known ILP datasets, four of which are considered to be standard ILP problems. These datasets were used for validation and justification of our concept and partial novel solutions that we propose, for setting the parameters and also for benchmarking with other state-of-the-art ILP algorithms.

In the benchmarking we compared the results of our algorithm to the results of the state-of-the-art miners presented in the literature. This comparison showed that the accuracy of models produced by our algorithm is comparable to the results of the other miners and for some problems it even outperforms these systems.

To conclude, the experiments showed the POEMS-based ILP learner is a competitive EA-based system that is able to solve ILP tasks while is not too specialized so that it would lose the EA advantages - simplicity and maximum domain independency of the implementation.

8 Problem-specific Actions

In this chapter we provide description of one of possible methods to extend the search scope of our system – utilization of problem-specific library refinements. The library contains clauses that describe frequent fragments that we discover prior to the run of the POEMS algorithm by the automated Gaston subgraph mining algorithm.

This extension is primarily motivated by improving the classification accuracy of the models. Utilization of additional set of problem-specific refinements can improve the search capability of our system by extending its radius by meaningful refinements - e.g. allow to bridge non-interesting plateaus or reach beyond range of the basic refinements by adding more complex specific sub-structures (e.g. benzene ring in chemical datasets). In the following sections we will first introduce a simple method to extend the set of actions for chemical datasets by automatically mined frequent fragments and analyze the effect of this extension on the quality of the final model. Then we will justify this approach by showing experimentally that it helps to increase accuracy of the final model.

8.1 Extending the POEMS Actions Set

When realizing the search for clauses by refinement-based approach, we came across an idea to extend the set of basic refinements with a library of larger clause "*fragments*" to help the searcher efficiently construct more complex clauses. To do this we can use one of the advantages of our POEMS based approach - the ease of potential extension of the set of actions by problem specific actions. We created an action that takes the clause fragments from the library and adds them to the actually modified working prototype clause.

Because in ILP we are dealing with relational data the natural option is to create the library by means of approach related to ILP – Graph Mining. Graph Mining (GM) (Cook and Holder, 2007) is a relatively recent and quickly developing branch of the data mining domain focusing on processing data that are structured as graphs (such as chemical compounds) – see Section 2.3.2 for more details. Because the expressivity of the graph representation is lower than that of FOL (e.g. no variables or functions can be used with graphs), GM can theoretically solve the task of frequent fragment mining faster than it would be solved in ILP. The condition is that the input data have form of a set of ground facts.

Using Graph Mining algorithm to build the library of pre-defined refinements offers following advantages:

- examples described in FOL by a set of facts can be easily converted to graph representation (see the next subsection);
- lower expressivity of graphs (compared to FOL) enables for faster mining of subgraph fragments than if the similar task would be solved by ILP techniques;
- current GM algorithms are optimized to efficiently and fast mine given graph database for all frequent subgraph fragments of defined property (minimal coverage and size);
- the process of GM is automated and requires no user input (except for the parameters); the output of the mining is set of all subgraph fragments of given property.

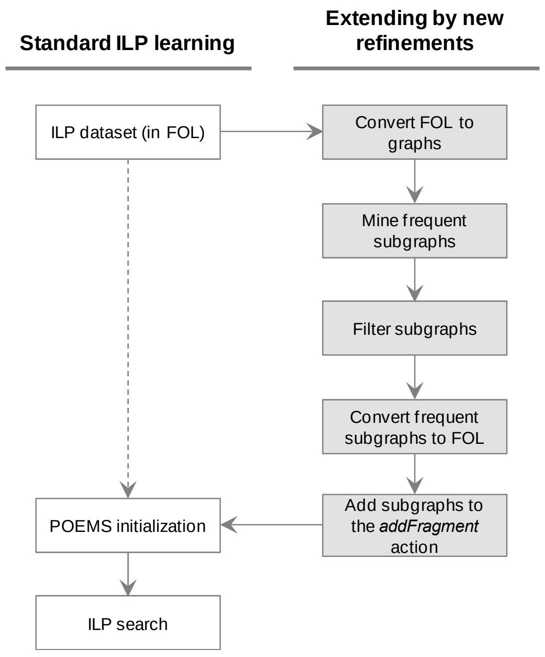


Figure 8.1 – Extending the POEMS refinement actions with frequent fragments

Building of the library happens prior to the learning phase within the following procedure (see also Figure 8.1):

1) *Converting FOL to graph representation.* To be able to use the graph mining algorithm the examples are converted from FOL to the representation of undirected labeled graphs. Because we were using this approach on chemical compounds datasets we experimented with two types of conversions:

– *full conversion* that preserves all relations defined by the predicates and
– *chemistry specific conversion* that transforms the set of facts into standard structures of chemical molecules.

An example of both conversion types is presented below in the next subsection.

2) *Frequent subgraph mining.* The search for small frequent subgraph fragments in the data happens by means of the Gaston frequent subgraph mining algorithm. Except for the mined dataset the algorithm takes minimally following two parameters as inputs. The two constraints the produced fragments should fulfill are: the maximal size of the fragment (count of nodes) and the minimal fragment coverage (number of examples it appears in). On the output it produces a list of fragments frequently appearing in the analyzed data. In our approach we used the Gaston algorithm (Nijssen and Kok, 2005) to automatically produce fragments for this additional library of refinements that can is then used by the POEMS during clause search. We used the Gaston implementation from the ParMol package (Meinl et al. 2006).
3) *Feature selection.* In cases when there are too many fragments their count can be reduced by methods of feature selection (Blum and Langley, 1997) and filtering out the fragments with lower classification power. Possible options are to sort them according to their evaluation by the entropy function (see Section 3.4.1.2) or use the entropy based gain ratio (Hall, 1998), measures commonly used as a splitting criterion in decision tree induction. Based on the sorting we can select the best *n* fragments with the highest value of the ratio to be used in the refinement library. In this work we did not use any special fragment filtering operation.
4) *Conversion of fragments to FOL.* Because we want to use the fragments as extension to the standard refinements we need to convert them from graphs back to FOL representation. This is a reverse step to the step 1.
5) *New specialization action.* For the purpose of adding the fragments from the library to the evolved clause we created a specific action *addFragment(F)*. When applied, this action selects one of the fragments from the library and adds it to the existing working phenotype clause. The action is parameterized by one parameter *F,* a number that defines the fragment to be added (the library where the fragments are stored has the form of an ordered list).

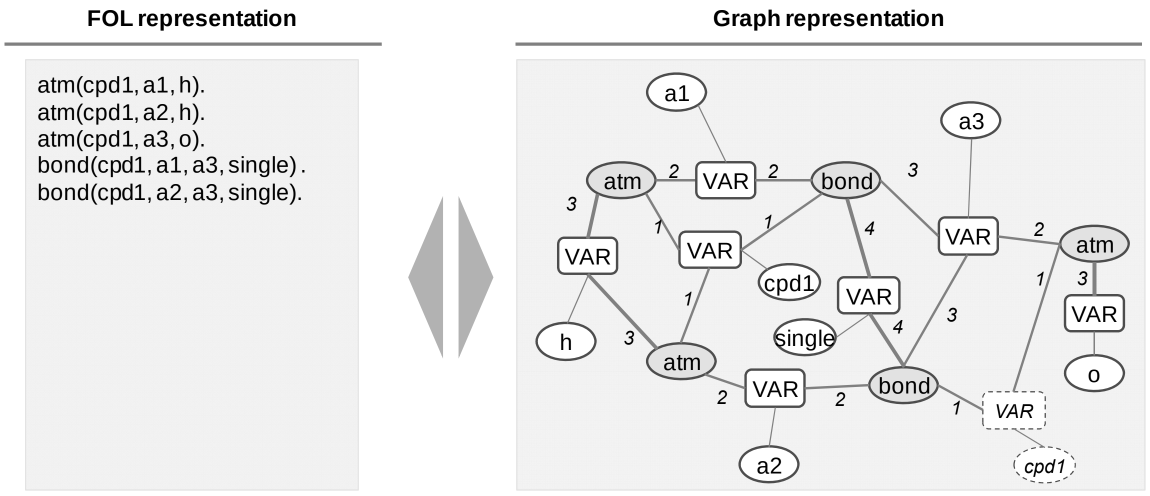
Gaston algorithm (GrAph/Sequence/Tree extraction) used in this work represents one of the latest development in the GM field. It uses efficiently organized complete search to mine given database of graphs for all subgraph fragments of defined property (minimal coverage and size). The method avoids cost intensive problems related to candidate generation and isomorphism tests by using a concept of own DFS code (in combination with lexicographic ordering) to canonize internal graph representation. DFS code of a graph is a string that is created by listing nodes and edges when depth-traversing the graph. In addition, it adopts an interesting hierarchical approach by first searching for frequent paths, then for trees and eventually for general graphs. This leads to faster and more efficient execution of the search process significantly reducing number of generated duplicates. Limitation of the algorithm is that it mines only continuous subgraphs (all nodes are connected to the rest of the subgraph).
The key for *Gaston* efficiency lies in mining the patterns within steps of increasing pattern complexity. Gaston first searches for all frequent paths, then all frequent free trees and finally cyclic

graphs. The first two types of patterns can be mined very quickly almost in polynomial time (Nijssen and Kok, 2005). The general graphs (patterns containing loops) for which the subgraph mining is extremely time demanding come at the end of the mining when the space of possible fragments was pruned by the mining for previous two fragment types. Also, the trees and paths serve as skeletons for the final full graphs. Gaston reaches the most efficient operation when mining graphs that do not contain many loops and is suitable especially for large molecular databases.
Two key parameters influence the result of the search – maximum fragment size and minimal fragment support (or coverage). These parameters affect not only the duration of the search but also number of fragments that are generated. The number of fragments generated rises enormously when lowering the fragment minimal support boundary. Therefore, efficient size limit (sometimes also combined with fragment filtering technique) needs to be employed.
Because of slightly unsatisfactory results of our algorithm over the chemical datasets of PTC we have extended the action sets for these datasets by actions that add molecular fragments that appear frequently in the training data.

8.2 Conversion between FOL and Graphs

The task of the conversion is to convert the examples (set of ground facts) from the FOL description to graph representation. Because we analyze molecular datasets (the *PTC* and *PTE* problems) we analyzed the following two options – full conversion that comes out of *binarisation of all the relations* and conversion *to chemical structures*. To show an example of conversion between FOL and graph description we will use a real situation of using this approach to solve the problem on molecular structure of the H_2O molecule (see Figure 8.2 and Figure 8.3 below).
Conversion based on full relation binarisation. The first option we have is to convert the FOL description into a graph so that every relation is preserved in the new representation. This conversion is depicted in Figure 8.2. In this method, each *n*-ary relation (predicate) is converted to binary relation that is then translated to the form of a graph. For this purpose we use nodes that represent argument variables. For example, the tertiary relation '*atm/3*' is in the graph represented by a subgraph – central note labeled as '*atm*' and three nodes (labeled as '*VAR*') each of which represents one argument of the relation. Order of the argument is given by the label of the edge that connects the variable node to the central relation node. Constants are treated as predicates with zero arity and as such they do not have any node representing the variable.
There are two types of edges in the graph:

- *Edges that connect variables with constants.* These edges are unlabeled and connect the specific constants to nodes representing variables that are further connected to predicates that use these constants.
- *Edges that connect relations to the variables.* Each predicate is connected with the number of variables that corresponds to its arity. These edges are labeled with the argument position in the predicate the variable corresponds to (e.g. variable that describes the first argument of predicate is connected to the predicate with edge labeled as '*1*').



```
atm(cpd1, a1, h).
atm(cpd1, a2, h).
atm(cpd1, a3, o).
bond(cpd1, a1, a3, single) .
bond(cpd1, a2, a3, single).
```

Figure 8.2 – Example of full general conversion between FOL and graph representation of H_2O molecule (the compound node '*cpd1*' split to 2 nodes for better transparency)

The advantages of this conversion are that we can keep most of the expressivity offered by ILP and can use it for any ILP dataset. The Figure 8.2 shows that potential subgraph pattern can describe even shared variables. However, from the figure it is clear that the corresponding graph representation is very complex (especially when we consider that it represents a simple molecule of water). Despite that there are only 3 atoms and 2 chemical bonds in the H_2O molecule the corresponding graph contains 19 nodes and more than 5 loops (the figure does not give exact picture about the structure as the node of the variable representing the compound ID '*cpd1*' split to 2 nodes for better transparency). Also we can see that the predicate nodes carry in this case low information and many structures using only these nodes will be certainly shared by all molecules.

Therefore, using this translation for molecules that contain tens or even hundreds of atoms would create graphs of large complexity. Actually, instead of simplifying the ILP search we would be blocked by extremely time-demanding frequent subgraph search because mining for subgraphs in graphs with large amount of common substructures and loops is extremely time demanding task. Therefore, we decided to decrease the level of expressivity and used another option in our building of library of refinements (see below).

Conversion to chemical structures. In this approach we use only the information that allows us to build graph that corresponds with the "standard" graphical description of given molecule. In this translation (depicted in Figure 8.3 below) the nodes of the graph represent atoms of the molecule and the edges of the graph represent the chemical bonds. No other information is used (e.g. no option to use shared variables etc.). We used only the labels of the atoms (in the *atm/3* predicate), labels of the bonds (in the *bond/4* predicate) and also ID's of the atoms to connect them with the bonds. The result is the basic molecular structure similarly to the one in Figure 8.3.

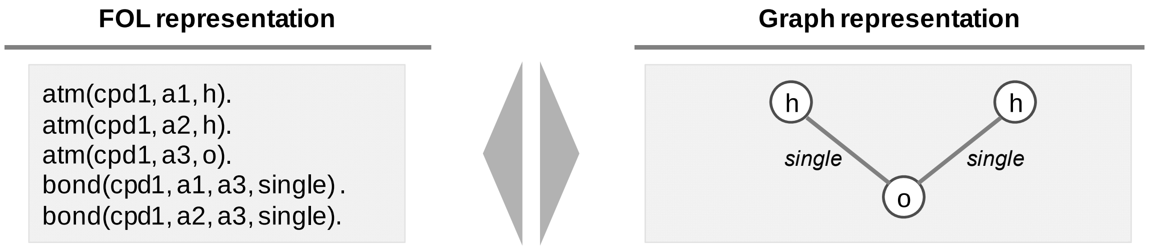


Figure 8.3 – Example of simplified conversion between FOL and graph representation

Comparing the two approaches we can see that the second option produces far less complicated graphs (yet with limited expressivity). The resulting graph is simple (3 nodes, 2 edges) and utilization of this approach for the graph mining task promises much faster solution. However, we needed to accept the price for this simplicity in the form of limited expressivity of the graph representation.

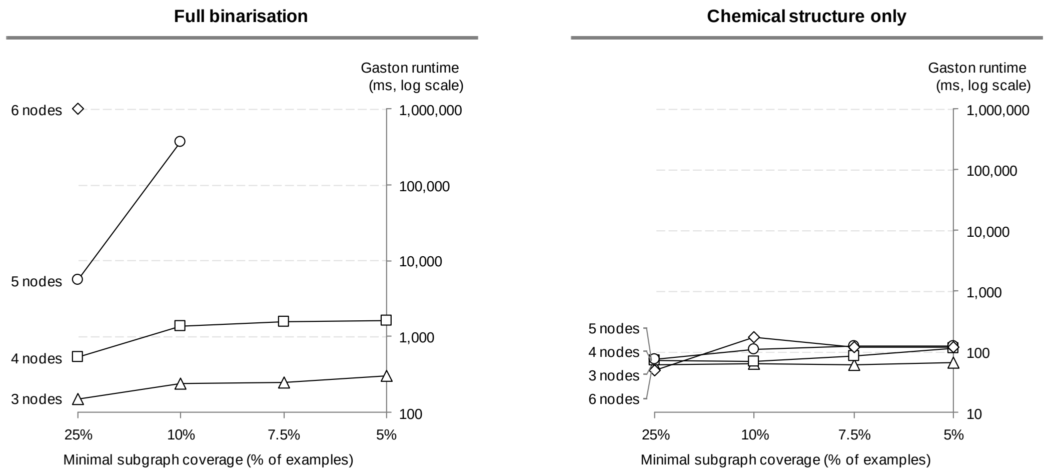


Figure 8.4 – Conversion method and runtime of the Gaston algorithm for different fragment sizes and minimal fragment coverages (showing results achieved in time less than 1.000 sec.)

To confirm our hypothesis that the graphs generated by the full conversion are too complex to be used for our purpose we tested effect of the conversion method and the related structure complexity on the runtime of Gaston algorithm. The dependency of the runtime of Gaston algorithm that produces the frequent fragments on the maximal subgraph size (in number of nodes) and minimal subgraph coverage (in % of examples) is in Figure 8.4. From the charts it is clear that the large amount of loops and large quantity of large common substructures (e.g. consider structures that consist only of atoms without element label connected only to the compound identifier node) clearly inhibits the function of the algorithm. For the maximal sizes of 5 and 6 nodes and minimal coverage of less than 10% we were even not able to obtain output before reaching the limit of 1.000 seconds.
On the contrary, when the conversion to simple chemical structures is used the Gaston algorithm is capable of very fast fragment mining almost regardless of the maximal subgraph size (at least if the

size lies in the range of few nodes only). Except for the complexity itself this is another reason why we use conversion to chemical structures in our approach.

8.3 Effect of Library Refinements on the Model Accuracy

In this subsection we will test the contribution of extending our algorithm with pre-defined "*library*" refinements. As described above, for this purpose we extend our action base (see Table 7.5) with a database of pre-defined clauses (fragments) and an action that adds the selected clause to the working prototype each time it is applied.

The first hypothesis to prove is that this extension of the basic POEMS action set will improve accuracy of the algorithm when used on the datasets *PTC* and *PTE*. Our motivation for selection of these specific datasets comes from two reasons:

1) the accuracy of the models built with standard action set is especially for the *PTC* datasets low (close to majority class share) and this extension might help to increase the models quality because it will help POEMS to generate longer but still meaningful clauses and
2) these datasets describe sets of chemical molecules and the transformation of these structures from the FOL description into graph form is simple and straightforward (we translated only the structures of the molecules) and we have the graph mining tool to quickly create the library.

For the experiments we used all four *PTC* datasets and also two variants of the *PTE* dataset – the basic one with all information from the original data (noted as *PTE*) and the constrained version (noted as PTE_{STR}) in which we use only the molecular structures and no other information (predicates *atom/5* and *bond/4* only, without other chemo-physical properties).

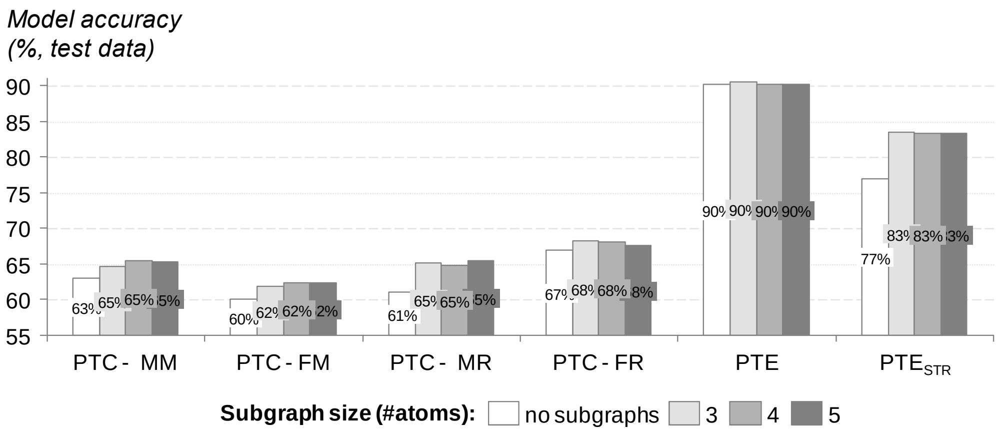


Figure 8.5 – Model accuracy for subgraph patterns of different sizes (*PTC* and *PTE* datasets, accuracy on test data

The second question we need to answer is the proper size of the fragments to use. While the larger fragments might have better classification power their size can lead to unacceptable runtimes of the algorithm.

The setup of the POEMS parameters is the same as determined by the previous experiments in Chapter 7. For each dataset we run five times the five-fold cross validation each time using different maximum fragment size constraint for the Gaston algorithm. We gradually increased the maximal fragment size in the sequence 3-4-5-6 atoms per fragment. This selection is motivated by the fact that translating molecular structures of this sizes back into FOL means obtaining clauses of the size of 5-7-9-11 literals (because each atom bond is also described by one literal).
The comparison of results obtained without using the fragments to the results reached with the fragments of various sizes are given in Figure 8.6. The results show that for the *PTC* datasets as well as for the *PTE*$_{STR}$ dataset (*PTE* dataset only with structural data on the molecules) the average accuracy of the models increased after the fragments were added. The extent of the increase depends on the dataset – the highest effect is observed for the *PTE*$_{STR}$ dataset (around +5%) while for the *PTC* datasets the increase is in the interval <1%; 4%>.

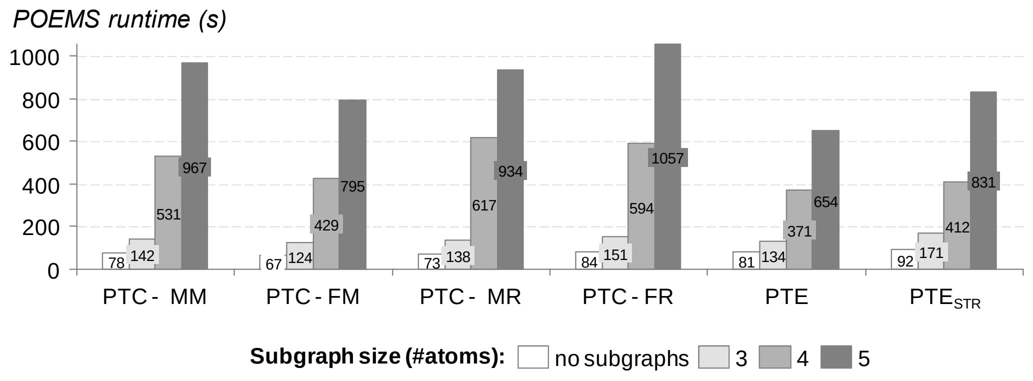


Figure 8.6 – POEMS runtimes for subgraph patterns of different sizes (*PTC* and *PTE* datasets, accuracy on test data)

Considering the size of the subgraph it seems that it is sufficient to use small fragments that contain only 3 atoms. Utilization of more complex fragments does not bring increase in the model accuracy.
However, as shown in Figure 8.6, the utilization of longer fragments leads to increase in the learning time while achieving similar accuracy as for the shorter fragments. The runtime increases steeply with the size of fragments and for fragments of 5 atoms the system runs approximately 10-times longer than without the fragments. This is clearly caused by tedious subsumption tests as fragments that contain 5 atoms actually contain at least 9 predicates (5 atom + 4 bond predicates). Adding such fragments to clause that already contains few predicates significantly slows down the search process.
Although the improvement on the *PTC* datasets is not large (few percentage points only), it helps the POEMS system to reach to the level of the best results achieved by the state-of-the-art ILP algorithms.
For the *PTE* dataset the most important information is included in the numerical properties of the atoms and compounds. Therefore, for the original *PTE* problem the experiment showed no improvement and the extension helped to improve the accuracy only for the version without numerical data (*PTE*$_{STR}$).

8.4 Conclusions

In this Chapter we described the utilization of problem-specific library refinements in our POEMS framework. The library contains clauses that describe frequent fragments that are automatically mined from the input data prior to the run of the POEMS algorithm. For this purpose we used the automated Gaston subgraph mining algorithm. Utilization of this additional set of problem-specific refinements improved the search capability of our system by extending its radius by meaningful refinements - e.g. allow to bridge non-interesting plateaus or reach beyond range of the basic refinements by adding more complex specific sub-structures (e.g. benzene ring in chemical datasets). This we justified in the second part of the Chapter where we analyzed the effect of this extension on the result of POEMS algorithm run. The experiments have shown that using this extension has positive impact on the classification accuracy of the final models.

9 Conclusions

In this thesis, we targeted the problem of learning classifying concepts in the relational domains (particularly Inductive Logic Programming) by means of Evolutionary Algorithms (EAs). We introduced a new approach to implementation of EAs into ILP that does not require development of special representation or search operators but it also utilizes full possibilities of ILP domain. The main reason for using the EAs in ILP is the complexity of the ILP tasks that actually require solving two NP-hard problems. Although this is not the first effort to use EAs in ILP, the main motivation was to overcome the bottlenecks of the existing EA-based ILP systems.

9.1 Summary of the thesis contribution

Our approach is based on the framework of the ***Prototype Optimization with Evolved Improvement Steps*** algorithm (POEMS) that represents a novel and suitable alternative for learning of logical hypotheses. In contrast to current approaches that focus on evolving the concepts we used the stochastic optimization process of EAs to iteratively adapt initial concepts (in the form of logical clauses) by means of context-sensitive clause refinement actions. Our algorithm works directly with clauses defined in FOL and alters them by means of sequences of ILP refinement operators that are searched for by evolutionary algorithm. Thereby, it modifies the initial concept so that it evolves towards the required conditions represented by clause evaluation function (e.g. the entropy function).
Such implementation does not demand development of any special problem-specific data structures and/or new complicated search operators like we can see among other similar systems. It also does not necessarily need any user-defined strong search and language bias (e.g. there is no necessary limit on clause form, length or on number of literals used) and offers an easy option to reach beyond FOL (e.g. by using numeric constraints). Direct access to the ILP refinement operators and direct operations with

clauses represented in FOL allows using full capabilities of ILP including incorporation of the domain knowledge. Moreover, utilization of the context-sensitivity in the refinement actions helps the algorithm to focus on syntactically correct concepts and to further refine them without creating abundant number of clauses that do not appear in the data.
The results of experiments on both synthetic and real-world data show that the proposed approach represents a competitive alternative to the current techniques. In experiments that used well-known non-trivial induction tasks, we have shown that this approach is able to construct relevant concepts both in artificial and real-world data and the obtained results were similar to the results of other evolutionary stochastic systems reported in literature. We also showed that this approach is able to handle and incorporate the domain knowledge and use it during the search process that is guided by standard genetic algorithm.

There are **five main contributions** of this thesis:

1) Adaptation of EA-based POEMS algorithm for the ILP domain.
2) Development of context-sensitive refinement actions
3) Development of algorithm for automated context induction
4) Extension of refinement actions with library based refinements produced by graph mining algorithm
5) Plateau facing technique based on complexity-preferring adjustment of the evaluation function.

Contribution 1 – the implementation of POEMS in ILP - introduces the *Prototype Optimization with Evolved Improvement Steps (POEMS)* algorithm into ILP. This allows also for the ILP tasks to directly use various optimizers that were originally developed to work with string arrays representation. Despite that we focused on using Evolutionary Algorithms (namely Genetic Algorithms), this approach can be easily extended with other similar optimization algorithms like Simulated Annealing, Evolution Strategies, various greedy searchers (e.g. GSAT) and many others. In contrary to standard application of EAs to ILP that requires development of specific problem representation and special search operators our approach can use the EA "*off the shelf*" even just with the standard EA operators. Using POEMS also allows us to solve the ILP problem while keeping the form of the solution (clause) unrestricted and using all possible clause refinements that ILP offers. Our approach therefore preserves to the maximum possible extent the advantage of the evolutionary optimization techniques – the domain independency – while fully using the possibilities of ILP.
This is important as especially finding out proper representation of ILP tasks so that they can be solved with EAs is difficult and usually is performed in a complicated way. Also the operator for recombination of general clauses is hard to define and so far has been developed only for restricted clause representations (e.g. bit-strings in system REGAL, trees in system GLPS or higher level strings in SIA01). Therefore, some of the EA-based ILP searchers (including the latest system ECL) only perform mutation.
Our POEMS approach links EA-based search with all standard ILP refinement operators (e.g. adding predicate to clause), but also allows to easily add potential non-standard operators (such as the one from Contribution 4). In common approaches that combine EA with ILP, this would disrupt the object-level GA in ILP. In our case this is solved by simple extension of the set of refinement actions.

The experiments showed that the proposed approach using POEMS with standard GA for solving ILP tasks is competitive with state-of the-art algorithms on the predictive accuracy. This holds both for comparison with standard algorithms (like Progol or Aleph) and also evolutionary-based ILP searchers like ECL.

Contribution 2 – the context-sensitive refinements - is critical for the efficient work of our refinement-action-based POEMS system in ILP domain. Utilization of context information during stochastic clause refinement efficiently enables the algorithm to focus on the syntactically and semantically correct clauses. Due to the enormous size of the ILP search space it is impossible to rely on the fact that necessary relations shall sometime "just appear" during the search. It is important to ensure that each literal of the constructed clause is properly related to the rest of the clause and that such relations are not immediately destroyed by the search mechanism.
To face this issue, when applying refinement operators we use the information about current state of the working prototype and constraints given by the structure of the analyzed data. In our approach, application of each operator (such as adding a literal or replacing a variable with a constant) is always bound to specific variable and its location within the whole working prototype clause and comes out of the structure of the analyzed dataset. In this way, the outcome of each refinement operator gets context dependent and we limit co-occurrences of unwanted literals or utilization of constants in undesired arguments. The information about the context restrictions is induced automatically prior to start of the learning process and requires no special user input (*contribution 3*).
The experiments we carried out proved usefulness of our context-based approach. Without context sensitivity it is hardly possible to randomly generate a clause with non-trivial coverage (clause that covers neither all nor any example) that would contain more than two predicates. On the contrary, for all the datasets used for evaluation the context-sensitivity of the actions allowed generating reasonable amount of random clauses (over 25% of all randomly generated clauses) that consist of even more than 6 predicates and still have non-trivial coverage.

Contribution 3 – the automated context induction - is one of the key elements for applicability of our algorithm to standard ILP problems. The context-sensitive actions showed that they can focus the search on meaningful parts of the ILP search space. However, they still need the context information as the input for their proper work. In this work we proposed an algorithm that runs prior to the start of the learning phase and automatically induces the context information from the data and saves it in a structure that can be easily used by the actions. We call this structure the *Graph of relations* in which we store the information about possible connections of literals (through shared variables) and possible utilization of constants. The fact that the user is not required to manually define the language bias by writing down the semantic and syntactic restrictions speeds up the whole process of solving ILP tasks and gives our system the user friendliness that is needed for using it to new problems. This is an advantage when compared to other systems that use similar principles of grammars. In these systems the grammars are usually user-defined. These manual interferences actually make the whole process of task solving with these systems time demanding.

Contribution 4 – the extension with library based refinements - shows the flexibility of our approach. The set of refinement actions used by the EA searcher can be easily extended by clause fragments supplied either by user or by automated pattern generator. In this work we automatically generate the interesting clause fragments by means of the frequent graph mining algorithm Gaston. The experiments showed that because this enhancement allows the searcher to reach more complex structures it helps to improve the final model accuracy (compared with the situation when this extension is not used).

Contribution 5 - complexity-preferring plateau facing technique - helps to solve problems with plateaus in the search space. Originally, the POEMS algorithm already includes several standard techniques to skip past the plateaus (utilization of EA searcher, possibility to shift the search to potential solution of the same quality as the best-so-far solution). However, the experiments with more complex datasets results showed that some other incentive is useful to support the search. We suggested realizing this by enriching the searcher's evaluation function with a complexity factor that – in contrast to standard methods – prefers more complex patterns to the smaller ones. This causes the searcher to prefer larger patterns when selecting among patterns with the same coverage and thus helps to overcome the plateau problem. An important point is that this complexity preference should be accompanied by clause generalization that is performed prior to adding the clause to the model. This solution is also suitable to EAs as with it we can directly influence the search process on all levels including the sequence optimization phase.
The experiments that were carried out in this thesis show that when combined with clause generalization this approach helps to improve the accuracy of the system when solving tasks with strong plateaus (compared with cases when complexity preference is not used).

When we compare the achievements of this thesis to the goals stated in the introductory motivation we can see that our system successfully fulfills all of them:

1) **Simple problem representation.** Our system uses the refinement-action-based interface to connect standard EAs (e.g. Genetic algorithm) with standard FOL engine that is usually used in ILP (e.g. SWI Prolog). This connection requires neither specialized problem representation nor a development of some specific search operators. Thereby, our implementation keeps one of the basic characteristics of EAs – the domain independency – while it still is able to use all approaches offered by ILP.
2) **Full utilization of ILP capabilities.** Our POEMS implementation operates directly with clauses defined in FOL and allows using any ILP refinement action. This includes also actions that work with numeric values like adding numeric constraints ($\leq$, $\geq$ and other) to the clause.
3) **Efficiently searching reasonable concepts.** Our method uses context-sensitive refinement actions to limit the search space on the reasonable clauses and to limit production of clauses that cover no examples. Thereby we can search reasonable concepts even without defining special EA operators or constraining the search by pre-defined clause templates.
4) **Refining discovered concepts.** The context sensitivity of the actions allows also for efficient refinement of the discovered clauses. This helps our refinement based ILP system to reach the good results in reasonable time.
5) **Easy utilization of additional domain knowledge.** We use the refinement-action-based interface to bridge the EAs with the FOL engine that is usually used to solve ILP tasks. This

principle allows using any of standard engines (e.g. SWI Prolog) and therefore there is no obstacle in including any existing problem domain knowledge into the search in a way it is used by standard ILP systems.

To conclude, we have presented a novel EA-based ILP system than previous EA-ILP systems that can use the any EA "*off the shelf*" to solve the ILP tasks without any need for development of special representation or EA operators. The effectiveness of the system was empirically demonstrated by benchmarking on standard ILP tasks. The experiments showed that our system produced models of comparable or better accuracy than various known existing ILP algorithms (current EA-based ILP searchers, kernel-based searchers and also standard ILP learners like Aleph or Progol).

9.2 Open Questions and Possibilities for Future Research

It was not possible for this thesis to answer all the questions related to the proposed implementation of the POEMS-based system to ILP. Several open questions and challenges still remain that could be targeted by the potential future research:

– Evolving theories instead of clauses - an interesting enhancement of our approach would be its extension to evolving theories at once (utilization of the *Pittsburgh approach*); while the set covering used in this work may not produce the optimal classification model (because it constructs it clause by clause in a greedy manner); using the whole model as the prototype solution and refining this model as a whole seems to be an interesting option that may push the possibilities of our system and quality of the produced models even further.

– POEMS algorithm parallelization - parallelism is a natural extension of any Evolutionary algorithm. The EA applications use parallel implementation to reduce the computational time. The main question is how to establish the communication between the individual threads of the learner so that in effect they efficiently search the space of clauses.

– Specific actions – the system that we introduced in this work can use any thinkable action to adjust the prototype clause; we have used only the standard ILP refinement actions (like adding or removing a predicate); however, development of special actions for the problem at hand might bring additional increase of accuracy of the final models.

– Automated methods for parameter tuning – the POEMS based system as introduced in this work takes only a few parameters, yet, the user of the system still must assign proper values to these parameters; development of self tuning algorithms for the set-up or correction of the model parameters (e.g. automated correction of the sequence length) would be another step towards faster runtime and user friendliness.

Appendix – The Implementation

In this section we will describe the basic structure of the implementation of POEMS algorithm for ILP we used in this work. We will describe its basic modular structure, focus on the most important modules and finalize the chapter by a section on description of a typical use-case of our system.

A.1 POEMS for ILP – Basic Structure

In this section we will introduce the structure of our system. The system consists of five basic blocks (see Figure A.1 for a schematic overview):

– *Iterative searcher* – this is a simple process that iteratively runs the algorithm for array optimization and manages the working prototype (exchanges the current one for the newly discovered one if the replacement conditions are met - the new prototype is better or equal to the current one). After given number of iterations or in case when the stopping criterion is met the iterative searcher terminates the search and returns last working prototype clause as the best clause discovered.

– *Array optimizer* – the function of the Array optimizer is to search for the array that encodes such sequence of actions that modifies the working clause into best possible clause. Generally, any sequence optimizer can be used in this block. In this work we experimented with Genetic Algorithm, Simulated Annealing, Evolution Strategies and simple Monte Carlo searcher. For most of the work we used the Genetic algorithm. Details on the implementation follow in the section below.

– *Array evaluator* – function of the Array evaluator is to evaluate given array produced by the Array optimizer. For this purpose it calls the mapper to obtain the working clause after the sequence has been applied to it. Then the evaluator calls the FOL coverage calculator to obtain number of examples that the clause covers. Finally, based on this coverage (and potentially also on the clause complexity) it calculates the value of the evaluation function and returns this number to the Array optimizer.

- *Mapper* – the mapper module translates the given chromosome into the sequence of refinement actions and immediately applies them to the working prototype. Result that is returned is the working clause modified by the sequence of actions that is coded by the chromosome. When processing parameters of each action, the *modulo* operator is used in order to ensure that parameters do not overflow maximal defined values. Both the *action id* parameters as well as the parameters for each action stay within required bounds. In effect, neither the numbers used for definition of *action ids* nor the action parameters need to be restricted or specifically transposed in some more complicated way.
- *FOL coverage calculator* – an FOL engine that is used to calculate coverage of given clause over the given example set. It takes the clause and for each example in the set it checks whether the example is covered by the clause. After experimenting with standard SWI Prolog we then switched to RESUMER2 engine due to its faster function. The result of the module's work is the number of positive and negative examples that are covered by the given clause.

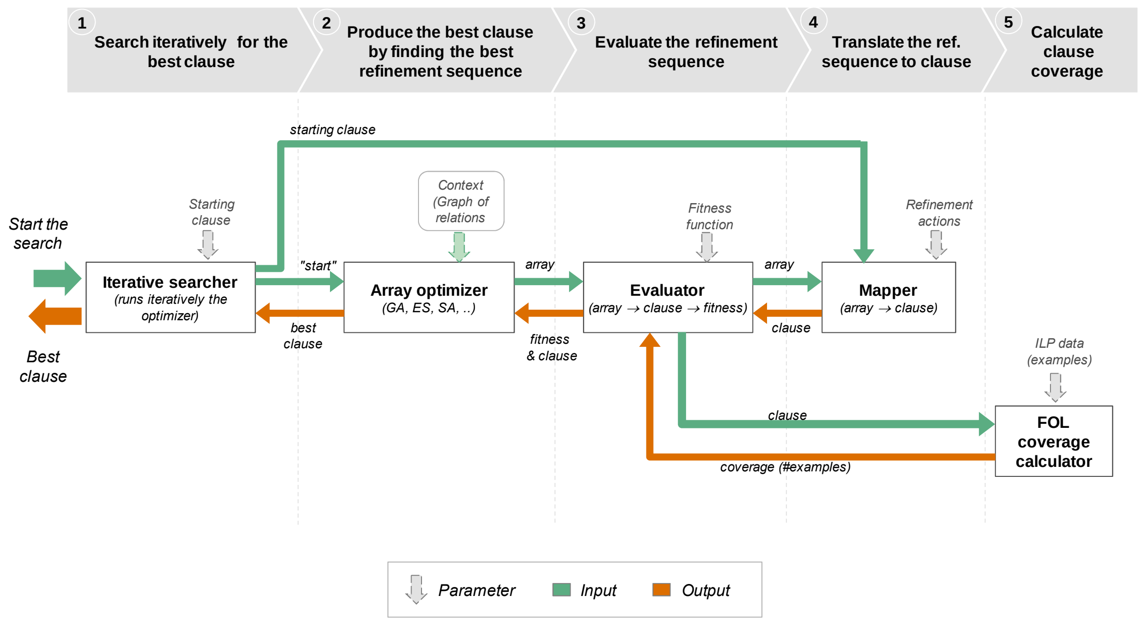

1 Search iteratively for the best clause
2 Produce the best clause by finding the best refinement sequence
3 Evaluate the refinement sequence
4 Translate the ref. sequence to clause
5 Calculate clause coverage
starting clause
Start the search
Starting clause
Context (Graph of relations)
Fitness function
Refinement actions
"start"
array
array
Iterative searcher (runs iteratively the optimizer)
Array optimizer (GA, ES, SA, ..)
Evaluator (array → clause → fitness)
Mapper (array → clause)
best clause
fitness & clause
clause
Best clause
ILP data (examples)
clause
FOL coverage calculator
coverage (#examples)
Parameter
Input
Output

Figure A.1 – Simplified structure of our implementation of POEMS for ILP

The *Iterative searcher*, *Evaluator* and the *Mapper* modules represent blocks with simple function that work almost without any special set-up. However, the *Array optimizer* and the *Coverage calculator* deserve more attention:

– *Array optimizer* - we used genetic algorithm as the optimizer and GAs can be set in many ways; basic details about our implementation of the GA are described in Section A.2;
– *Coverage calculator* - we used non-standard clause sorting and buffering approach in the module of FOL engine; details on our approach as well as its basic experimental evaluation are given in Section A.3.

A.2 Implementation of GA used in POEMS

Theoretically, any optimization algorithm can be used in POEMS to search for the best sequence of adaptations. The standard genetic algorithm as introduced in the previous chapter is not the only option of search algorithm that can be used within POEMS. Because we work with individuals that are represented in the form of arrays, the POEMS framework can easily use any other standard array optimizing algorithm.

The algorithm we used for array optimization in this work is the genetic algorithm (GA). If we use the genetic algorithm (GA) in POEMS, as we did in this work, we can easily use the standard blind crossover and mutation operators. This is an advantage as we do not necessarily need to develop own specialized operators.

To describe how we implemented the GA in this work we can use the structure of seven standard components used by standard GA framework:

1) evolutionary model (generational, steady state);
2) representation;
3) fitness function;
4) selection operators (parent selection, survivor selection);
5) variation operators (mutation and recombination);
6) initialization procedure;
7) stopping criterion.

Following this structure, in our work we used following GA set-up (values of the individual parameters were set after ~20 runs of the algorithm over each dataset to reflect the values with which the algorithm achieved best results):

Evolutionary model. Generational evolutionary model was used. In this approach the optimizer keeps two populations – one that is used as a source population from which it selects parental chromosomes and second population into which it copies the offsprings. When the size of the second population reaches the maximum size limit, the algorithm discards the source population and uses the second population as the new source population. The size of population in our implementation was 50 individuals, the maximal number of generations was set to 50.

Representation. Simple linear array representation was used. The array consisted of codons, each codon contained 4 numbers (bytes). The first number was treated as the action identifier, the remaining three were used as action parameters. In case when the action did not require all of the three parameters, the excess parameters were skipped (not used).

Fitness function. The selection of the fitness function was described in Section 7.3.2. From the available functions, entropy and novelty were used for the analyzed ILP tasks.

Selection Operator. Tournament selection operator was used for parent selection in our GA. Tournament is a useful and robust selection mechanism commonly used by genetic algorithms (Goldberg, 1989). The selection pressure of tournament selection directly varies with the tournament size - the more competitors, the higher the resulting selection pressure is. We used four competitors from which we select two best contestants to serve as the parents for the offsprings. No survivor selection operators are needed in the generational evolutionary model.
Recombination Operator. The one-point crossover operator (Goldberg, 1989) is used in this work. It generates child chromosomes by splitting each parent into two parts and swapping the tails. The probability of cross-over was set to p_{xOver} = 0.8. This means that in 80% of cases (random decision prior to the crossover) the crossover is carried out and the offsprings carry one half from each parent. In the remaining 20% of cases each offspring is simply a copy of one parent. By this mechanism the parents have 20% probability they will survive and be copied to the next generation.
Mutation Operator. A simple blind random uniform single gene-modifying operator was used. It can change any gene (byte) in the chromosome - either the action type or the parameters of the action. Each gene of the chromosome had the same 10% probability to be replaced with random number (p_{Mut}=0.1)
Initialization procedure. Fully random initialization was used. At each start of the algorithm run, the initial population was generated fully at random (including selecting random chromosome lengths from the given range – see Section 7.3.3.1 on the chromosome length selection).
Stopping criterion. The only stopping criterion used was reaching maximum numbers of generations.

A.3 Coverage Calculations

In this subsection we want to test the efficiency of using the clause predicate sorting and coverage buffer to shorten the time needed for coverage calculations. The main motivation is that for the clauses that have been already tested during the search we do not want to run expensive coverage calculations over the whole dataset again.
When analyzing methods how to decrease the time needed for the clause evaluation we realized that there are many clauses that our refinement based search tends to produce several times. This is caused by the fact that our algorithm performs the search by stochastically adjusting the working prototype clause. Random search in the neighborhood of the prototype clause naturally leads to the fact that many clauses are produced more than once. However, the same clause can take many forms as the predicates can be ordered randomly and so can be the variables. The question therefore was how to face this issue of redundant coverage calculations.
Our main hypothesis is that the lexicographical sorting of clause predicates as introduced in Section 6.3.3.2 is sufficient to filter large majority of duplicite clauses that are naturally generated during the search of our POEMS based algorithm and we do not need to implement other more sophisticated filtering method.
The second thing we want to show in this subsection is that measuring of unique clauses generated during the search is good approximation of the time needed for the run of the algorithm (motivation for this is that we are using several machines with different parameters to test our algorithm and the algorithm run times in seconds would not be comparable).

A.3.1 Buffering the Clause Coverages

We use a simple technique to optimize the evaluation process by buffering clauses and their coverages (see Algorithm A.1). The predicates of the clause are ordered which enables simple checking for clause equality by performing text comparisons. Each time a clause is passed to the evaluating engine we first test whether the coverage has not been already calculated. In order to avoid expensive clause equality tests when searching for the controlled clause in the buffer we adopt a simple solution – we order the literals of each clause by an ordering '$\prec_L$' that combines the literal name, arity (number of arguments), constants and variables used in the literal (see definition of $\prec_L$ in the Section 6.3.3.2).

Algorithm A.1 – Chromosome evaluation with the clause buffer

```
function evaluateChromosome
   Input: working clause C, chromosome CHR, evaluation function EvalFcn, FOL
            engine FolEngine, set of examples E, coverage buffer Buffer
   Output: evaluation of the chromosome ClauseQuality

   NewClause = translateChromosome(CHR, C)
   OrderedClause = orderClauseLiterals(NewClause)
   ClauseCoverage = getCoverageFromBuffer(OrderedClause, Buffer)
   if(ClauseCoverage != null)
      ClauseQuality = evaluateClause(ClauseCoverage, EvalFcn)
      return ClauseQuality
   else
      ClauseCoverage = calculateCoverage(OrderedClause, E, FolEngine)
      saveCoverageToBuffer(OrderedClause, ClauseCoverage, Buffer)
      ClauseQuality = evaluateClause(ClauseCoverage, EvalFcn)
      return ClauseQuality
```

A.3.1 Efficiency of Clause Predicate Ordering

In this subsection we describe the experiment we performed to analyze the efficiency of detecting clause duplicity. In the experiment we supplemented the clause comparison technique we are using in this work by full subsumption test and measured the share of clauses that are detected to be new by simple lexicographical comparison but they have been actually generated before and just take different form after the sorting procedure. For this purpose we have generated 10,000 random sequences of different action lengths, called the coverage evaluation procedure and measured the duplicities not detected by the buffer.

The results of this experiment are given in Figure A.2. The chart shows that for the *PTE*, *PTC MM* and the *Alzheimer memory* datasets the lexicographical comparison of sorted clauses discovered almost all of the duplicities. Slightly different situation is for the *Trains* and *CAD* datasets, however, also in these cases the share of duplicities did not reach over 2% (*Trains*) or 7% (*CAD*) of the total amount of clauses generated. The CAD and Trains dataset seem to appear worse in this regard because –

compared to the other three ILP problems – in these datasets there exists much more complex clause patterns that are shared by all or most of the examples.

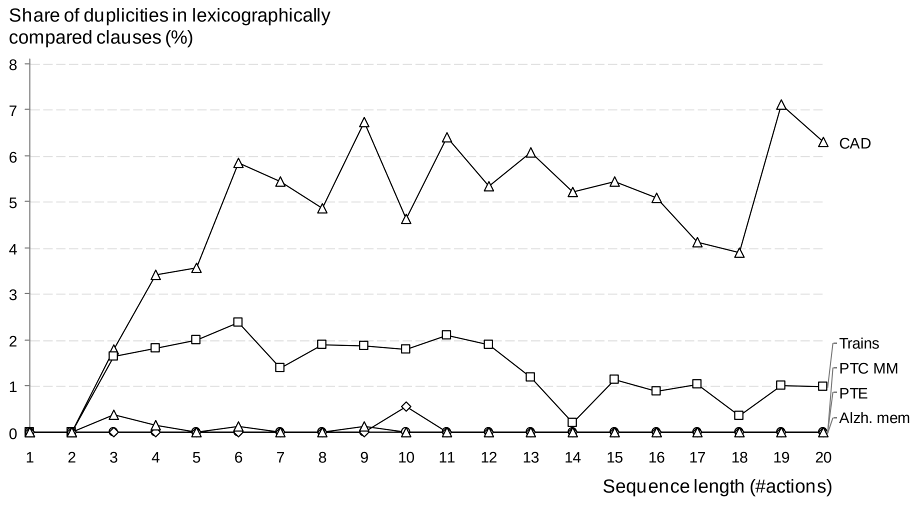


Figure A.2 – Share of non-detected duplicities when using simple lexicographic clause equality test

In the second experiment we tested the effect of using the buffer with different method of evaluating clause equality. We compared three approaches:

1) the suggested approach that sorts literals of the clause and the compares the text form of the clauses
2) approach in which we save and compare the clauses in the form they were produced (not sorted literals) and
3) approach in which we sort the literals but also use exact subsumption test to check for clause equality (used in case when the search through the simple text comparison does not detect the equality).

With each of these examples we run five-fold cross validation for each of the datasets we use for our experiments. The average runtime values per one fold for each of the methods are in Figure A.3.

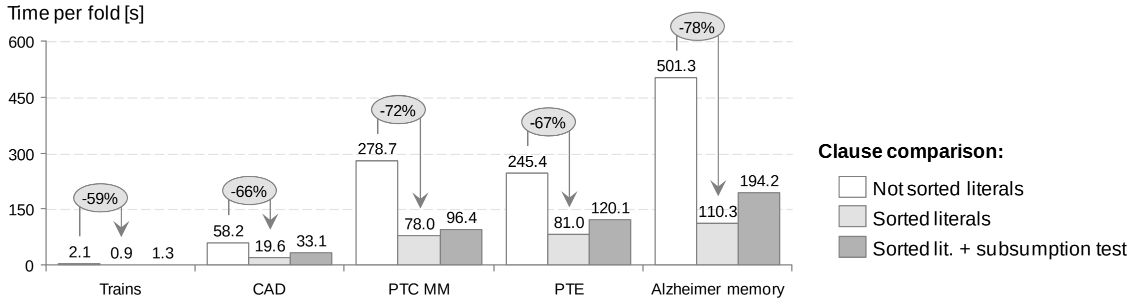


Figure A.3– Average time per one fold when different methods of clause comparison are used

Results of the experiments clearly prove that our approach leads to the shortest runtimes. When we compare our suggested approach with the approach that does not use literal sorting we can see that the time savings are from 59% (for the *Trains* dataset) to 78% (*Alzheimer memory* dataset). On the other hand we also see that reaching full perfection and checking for clause equality ends up in longer run times than using our approach with the risk that for a few percent of the clauses the coverage will be calculated more than once. These two experiments justify our hypothesis that using text comparison over clauses with sorted literals is good approach to reach sufficient time savings in the algorithm run.

A.3.2 Time Needed for Ordered Clauses Evaluation

The second issue we wanted to prove is that measuring of unique clauses generated during the search is a good approximation of the space searched and also of the time needed for the run of the algorithm. Additionally, we want to justify that the number of unique clauses generated during the search gives better approximation of the runtime than the total number of all clauses generated during the search. This is due to the fact that we use clause coverage buffer (as described in the previous sub-section).
To prove this we designed following experiment. We gathered the results of 250runs (10 experiments that were produced in Chapter 7 each of which used the five times five-fold cross validation). All the runs were produced with the same set-up as given in Table A.1 below.

Table A.1 – System setup for assessing the time dependency

POEMS set-up	5 iterations, model size unlimited, sequence sizes and minimal clause coverage acc. to dataset
GA set-up	recombination: 1-point crossover (p_{xOver} = 0.8), mutation: uniform simple mutation (p_{Mut} = 0.1), population size: 50, generations count: 50, selection: tournament (2 of 4)

For each of the folds we plotted the runtimes versus number of unique clauses searched and number of total clauses searched (including the duplicities). We also used linear regression and the R-squared coefficient to estimate the intensity of the dependency between the runtime and both numbers of generated clauses. The results are given on the charts below (Figure A.6 and Figure A.4).

Both the scatter plots and the R-square indexes presented in the figures show that the dependency between the number of unique clauses and the runtime is much stronger than the dependency between the total number of clauses searched and the runtime. Therefore, we will use the number of unique clauses as an approximation of the search complexity.

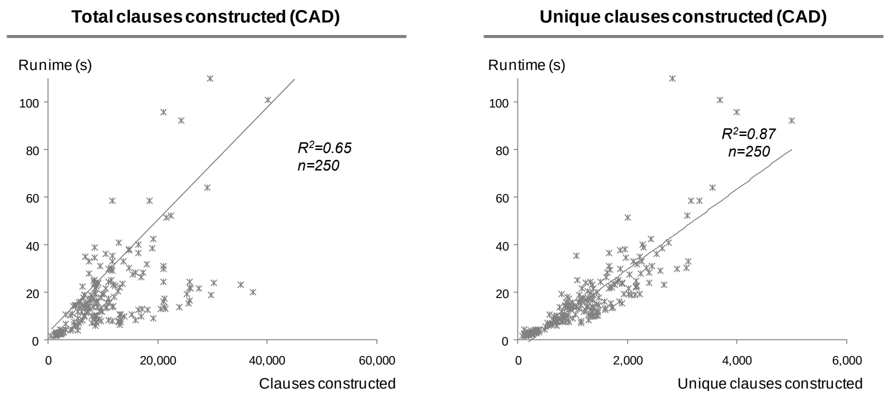


Figure A.4 – Runtime vs. total clauses constructed and unique clauses constructed (10 experiments, each 5 times five-fold cross validation, *CAD* dataset)

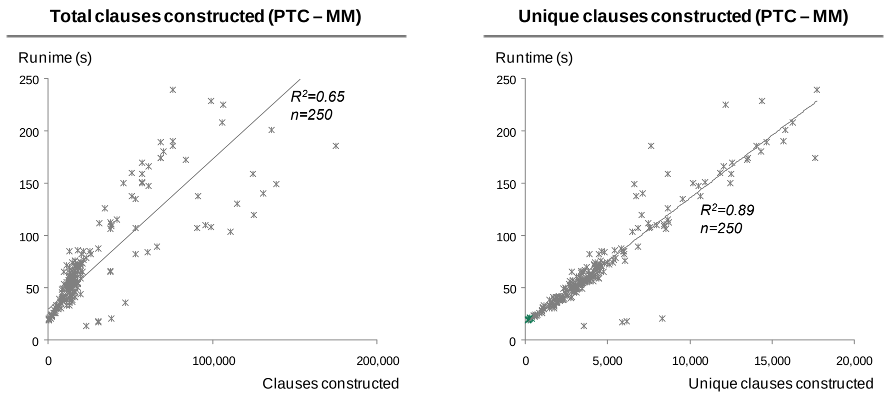


Figure A.5 – Runtime vs. total clauses constructed and unique clauses constructed (10 experiments, each 5 times five-fold cross validation, *PTC MM* dataset)

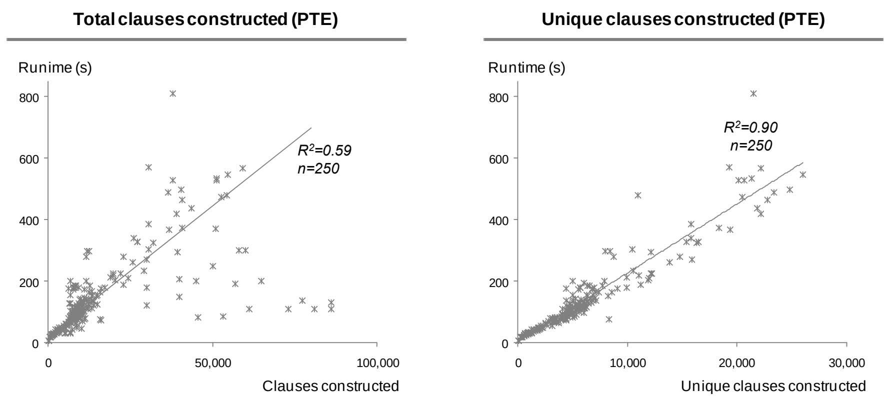


Figure A.6 – Runtime vs. total clauses constructed and unique clauses constructed (10 experiments, each 5 times five-fold cross validation *PTE* dataset)

A.3.3 Clause Evaluation - Conclusions

In this subsection we analyzed the effect of the simple sorting procedure to avoid duplicite coverage calculations for the clauses that have been already evaluated during the search. According to the experiments our approach seems to be sufficient – we see that undetected duplicities appear only in the case of *CAD* (~7% of clauses) and *Trains* data (~0-2% of clauses). Considering the simplicity of our solution, we think that the small share of duplicite calculations in the case of CAD dataset and 0% increase in runtime for other datasets is acceptable and we will not try to implement any more complex and sophisticated solution.
We also examined the relation of runtime and number of unique clauses and number of total clauses generated during the search. We showed that the dependency of runtime in number of unique clauses is higher than on total number of clauses (due to the clause coverage buffering). Therefore, the number of unique clauses gives reasonable approximation of the runtime and shows better estimate than the total number of clauses.

A.4 Using Our System – Typical Workflow

To give an idea about how our system is utilized for building the classification models, we would like to present here a brief overview about the standard workflow in which we use the system. This standard workflow that is employed when our POEMS-based system is used to analyze an ILP task can be summarized into 7 steps:

1) *Prepare the ILP data for the FOL engine.* In the first step we need to process the data given by the ILP task (E^+, E and background knowledge$^-$) and parse them to a form that is understood by the FOL engine (either Prolog or Resumer2). This is a standard data preparation step that cannot be avoided when using any of existing ILP systems.
2) *Create list of predicates to use (optional).* In this step the user has the possibility to select which predicates he wants to use for clause construction. If no specification is given the system uses all the predicates available in the data.

3) *Create list of predicate arguments where constants shall be used (optional).* In this step the user has the possibility to define which predicates and which arguments shall be used for the learning. In standard ILP datasets, some arguments of some predicates are usually reserved to define entities (e.g. compound IDs, atom IDs etc.). Using constants on places of these arguments cannot bring any value added to the learning as the constants carry no meaning and often are unique for the whole dataset (e.g. in '*train(t1), hasCar(t1,c1)*' the constants '*t1*' and '*c1*' are used to define entities and should not be used during search for patterns). The user has the option to prohibit constants to be used in some predicate arguments. If no specification is given the system will use constants that appear more than once in the dataset (this threshold can also be changed).
4) *Create list of predicate arguments where variables can be shared (optional).* Similarly to the previous point user has also the option to select which arguments of which predicates he wants to use for variable sharing (e.g. '*train(X), hasCar(X,Y)*' shares the variable '*X*'). If no input is given, the system will allow sharing variables between all predicate arguments that share at least one constant (e.g. '*train(t1), hasCar(t1,c1)*' shares the constant '*t1*').
5) *Define the start phenotype (optional).* If user does not specify the starting clause empty clause will be used.
6) *Set-up the search parameters (optional).* In this step user needs to set-up the parameters that were analyzed in this thesis. He needs to:
    a) select the optimization algorithm (GA, SA, etc.; optional step, GA is used if not defined otherwise),
    b) define the set of actions (optional - if not defined, standard set of ILP refinement actions that are available will be used),
    c) set-up the search restrictions (limit on number of iterations, sequence lengths, model size, minimal clause coverage etc.),
    d) select the method of evaluation (select the evaluation function, define the number of cross-validation folds or the training/testing data split sizes).

    The step of setting the system parameters generally cannot be avoided with any ILP system. In our case, the user should specify at least the search restrictions (numeric parameters of the system – step 6a) but generic values are used if the user does not wish to redefine the parameters.
7) *Build the model.* After all the previous steps have been finished (or skipped) the algorithm is ready to load the data and start creating the model. After starting the search, no further input is needed here.

A.5 Conclusions

In the Appendix we described the basic modular structure of our implementation of the POEMS-based system for ILP. We described each module and then focused on two modules that require higher attention – on the *Array optimizer* because it uses genetic algorithm that offers many possibilities of implementation and on the *FOL Coverage calculator* because we use non-standard clause coverage buffering procedure with this module.

After describing our implementation of the GA we focused on the coverage calculation optimization. The main issue was avoiding the redundant coverage calculations with clauses that have been already produced during the search. To tackle this we used simple clause predicate sorting method. The experiments showed that our approach is capable of removing most of the duplicities in clause description. We also showed that when combined with coverage buffering it shortens the search time by approximately 60-75% which justifies its utilization. Finally, we showed a typical use-case in which we use our system to analyze the ILP data.

References

Anglano C., Giordana A., Bello G. L., and Saitta L. (1998). *An Experimental Evaluation Of Coevolutive Concept Learning*. In Proceedings of the 15th Intl. Conf. on Machine Learning, Morgan Kaufmann, pp. 19-27.

Anglano C. and Botta M. (2002). *NOW G-Net: learning classification programs on networks of workstations.* In Trans. Evol. Comp 6, 5 (October 2002), 463-480.

Blockeel H. and De Raedt L. (1998). *Top-down induction of first-order logical decision trees.* Artificial Intelligence 101(1-2), pp. 285–297.

Blockeel H., Dehaspe L., Demoen B., Janssens G., Ramon J. and Vandecasteele H. (2002). *Improving the efficiency of inductive logic programming through the use of query packs*. In Journal of AI Research, 16, pp.135-166.

Blum A. L. and Langley P. (1997). *Selection of relevant features and examples in machine learning.* In Artificial Intelligence 97 (1-2), pp. 245-271.

Bongard, M. (1970). *Pattern Recognition*, Hayden Book Company.

Borgelt, C., Berthold, M.R. (2002). *Mining Molecular Fragments: Finding Relevant Substructures of Molecules*. In: Proc. IEEE Int'l Conf. on Data Mining ICDM, pp.51–58. Japan

Camacho R. (2003). *As Lazy as It Can Be.* In P. Doherty, B. Tassen, P. Ala-Siuru, and B. Mayoh, editors, The Eighth Scandinavian Conference on Artificial Intelligence (SCAI'03), pp. 47–58. Bergen, Norway.

Campbell D.T. (1965). *Variation and selective retention in socio-cultural evolution*. In: Herbert R. Barringer, George I. Blanksten and Raymond W. Mack (Eds.), Social change in developing areas: A reinterpretation of evolutionary theory, pp. 19–49. Cambridge, Mass.: Schenkman

Cleary J. G., Legg S., and Witten I. H. (1996). *An MDL Estimate Of The Significance Of Rules.* In Proceedings of ISIS: Information, Statistics, and Induction in Science.

Cohen W.W. (1994). *Grammatically Biased Learning: Learning Logic Programs Using An Explicit Antecedent Description Language*, in Artificial Intelligence, 68(2), pp.303-366.

Cook D. J. and Holder L. B., editors (2007). *Mining Graph Data*. Wiley, ISBN: 978–0–471–73190–0

Costa V.S., Srinivasan A., Camacho R., Blockeel H., Demoen B., Janssens G., Van Laer E., Cussens J., and Frisch A. (2002). *Query Transformations for Improving the Efficiency of ILP Systems*. In Journal of Machine Learning Research, vol. 4, pp.491-505.

Cramer, N.L. (1985). *A representation for the Adaptive Generation of Simple Sequential Programs.* In Proceedings of an International Conference on Genetic Algorithms and the Applications, Grefenstette, John J. (ed.), Carnegie Mellon University

Černý, V. (1985). *Thermodynamical approach to the traveling salesman problem: An efficient simulation algorithm*. Journal of Optimization Theory and Applications 45, pp. 41–51.

Dantsin E., Eiter T., Gottlob G. and Voronkov A. (2001). *Complexity and Expressive Power Of Logic Programming*, ACM Computing Surveys, 33, pp. 374-425.

Davis, L., ed. (1991). *Handbook of Genetic Algorithms*, NY: Van Nostrand Reinhold.

De Raedt L. and Blockeel H. (1997). *Using logical decision trees for clustering*. In N. Lavrac and S. Dzeroski (eds.), Proceedings of the Seventh International Workshop on Inductive Logic Programming, Lecture Notes in Artificial Intelligence 1297, pp. 133–140. Springer-Verlag, 1997.

De Raedt L. and Dehaspe L. (1997). *Clausal discovery*. Machine Learning, 26(2/3): pp. 99–146.

De Raedt L. (1998). *Attribute-Value Learning Versus Inductive Logic Programming: The Missing Links (extended abstract)*, in LNAI 1446: Proceedings of the 8th International Conference on Inductive Logic Programming. Springer Verlag.

De Raedt L. (2008). Logical and Relational Learning. Springer Verlag.

Dietterich T.G. (1988). Approximate Statistical Tests For Comparing Supervised Classification Learning Algorithms. *Neural Computation*, 10(7), pp. 1895–1924.

Divina F. (2006). *Evolutionary Concept Learning in First Order Logic: An Overview*, in AI Communications. IOS Press, Volume 19, Number 1, pp. 13-33.

Dzeroski S. and Lavrac N. (eds.). (2001). *Relational Data Mining*. Springer.

Flach P. A. and Lavrac N. (2002). *Learning in Clausal Logic: A Perspective on Inductive Logic Programming*. In Computational Logic: Logic Programming and Beyond, Essays in Honour of Robert A. Kowalski, Part I, Antonis C. Kakas and Fariba Sadri (Eds.). pp. 437-471. Springer-Verlag, London, UK.

Fogel L.J., Owens A.J., Walsh M.J. (1966) *Artificial intelligence through simulated evolution*. Wiley, New York

Fonseca N. A., Rocha R., Camacho R., and Santos Costa V. (2007*). ILP: Compute Once, Reuse Often.* In Workshop on Multi-Relational Data Mining, pp. 34-45.

Fonseca N.A., Costa V.S., Rocha R., Camacho R., and Silva F. (2009). *Improving the Efficiency of Inductive Logic Programming Systems*. Software – Practice and Experience, 39(2), pp. 189–219.

Fröhlich H, Wegner JK, Sieker F, Zell A (2005).*Optimal assignment kernels for attributed molecular graphs*. In: Proceedings of the 22nd international conference on Machine learning. ACM Press, pp. 225–232.

Fuernkranz J. (1999). *Separate-and-Conquer Rule Learning*. In Artif. Intell. Rev. 13(1), pp. 3–54.

Giordana A. and Neri F. (1996). *Search-Intensive Concept Induction*, in Evolutionary Computation Journal, 3, pp. 375-416.

Goldberg D. E. (1989). *Genetic Algorithms For Search, Optimization And Machine Learning*. Addison-Wesley.

Gottlob, G., Leone, N., Scarcello, F. (1997). *On the complexity of some inductive logic programming problems.* In Proc. of the 7th Int. Workshop on ILP. Volume 1297 of LNAI. pp. 17–32.

Hall M.A. and Smith L.A. (1998). *Practical feature subset selection for machine learning.* In Proceedings of the 21st Australian Computer Science Conference, pp. 181–191.

Hekanaho J. (1998). *DOGMA: A GA-Based Relational Learner*, in Proceedings of Inductive Logic Programming, 8th International Workshop.

Helma, C., King, R. D., Kramer, S., and Srinivasan, A. (2001). *The Predictive Toxicology Challenge 2000-2001*. In Bioinformatics, 17, pp. 107-108.

Helma C., Cramer T., Kramer S., de Raedt L. (2004). *Data Mining and Machine Learning Techniques for the Identification of Mutagenicity Inducing Substructures and Structure Activity Relationships of Non-congeneric Compounds*, J. Chem. Inf. Comput. Sci., (44), pp. 1402-1411. DOI 10.1021/ci034254q

Holland, J. H. (1975). *Adaptation in Natural and Artificial Systems*, Ann Arbor MI: Univ of Michigan Press.

Huan J., Wang W., Prins J. (2003). *Efficient Mining of Frequent Subgraphs in the Presence of Isomorphism*. In ICDM, p. 549, Third IEEE International Conference on Data Mining (ICDM'03)

Huan J., Wang W., Bandyopadhyay D., Snoeyink J., Prins J., and Tropsha A. (2004). *Mining family specific residue packing patterns from protein structure graphs.* In RECOMB, pp. 308–315.

Huynh, T. N. (2011). *Improving the Accuracy and Scalability of Discriminative Learning Methods for Markov Logic Networks*, Dissertation Thesis, University of Texas at Austin.

Inokuchi A., T., Okada T., Motoda H. (2001). *Applying the Apriori-based Graph Mining Method to Mutagenesis Data Analysis*, In Journal of Computer Aided Chemistry, vol. 2, pp.87-92

Inokuchi A., Washio T., and Motoda H. (2003). *Complete mining of frequent patterns from graphs.* In Mining graph data, Machine Learning, 50, pp.321-354.

Inokuchi A. (2004) *Mining Generalized Substructures from a Set of Labeled Graphs*, Proceedings of the Fourth IEEE International Conference on Data Mining (ICDM'04), pp. 415 – 418, IEEE Computer Society

J. Rissanen. (1978). *Modeling By Shortest Data Description*. *Automatica*, vol. 14, pp.465 – 471.

Karalic A. and Bratko I. (1997). *First-order regression*. Machine Learning, 26(2/3), pp.147–176.

Karr C. and Freeman L. M. (eds.). (1999). *Industrial Applications of Genetic Algorithms*, CRC Press.

Kautz H.A., Selman B. (1996). *Pushing the Envelope: Planning, Propositional Logic and Stochastic Search*. In Proceedings of the 13th National Conference on Artificial Intelligence (AAAI'96), pp. 1194-1201.

Kennedy C. J., Giraud-Carrier C. (1999). *An Evolutionary Approach to Concept Learning with Structured Data*. In Proceedings of the fourth International Conference on Artificial Neural Networks and Genetic Algorithms, pp. 1–6.

Ketkar N., Holder L. and Cook D. (2005). *Comparison of graph-based and logic-based multi-relational data mining*. SIGKDD Explor. Newsl. 7, 2 (December 2005), pp. 64-71.

Kietz J. U. and Lobbe M. (1994). An *efficient subsumption algorithm for Inductive Logic Programming.* In Proc. 11th Int. Conf. on Machine Learning, pp. 130-138. Morgan Kaufmann.

King R, Sternberg M, Srinivasan A. *Relating Chemical Activity to Structure: An Examination of ILP Successes.* New Generation Computing. 1995; 13(3&4): 411-433.

King, R. D., Srinivasan A., Dehaspe L. (2001). *Wamr: a data mining tool for chemical data*, J. Comput.-Aid. Mol. Des., 15, pp. 173-181.

Kirkpatrick, S.; Gelatt, C. D.; Vecchi, M. P. (1983). *Optimization by Simulated Annealing*. In Science 220 (4598): pp. 671–680.

Koza, J. R. (1992). *Genetic Programming*, MA: MIT Press.

Kubalik, J. (2009). *Solving Multiple Sequence Alignment Problem Using Prototype Optimization with Evolved Improvement Steps.* In Proceedings of the ICANNGA 2009, Kuopio, Finland, 23–25, April.

Kubalik, J. (2011). *Evolutionary-Based Iterative Local Search Algorithm for the Shortest Common Supersequence Problem.* In Proceedings of the 13th Annual Conference on Genetic and Evolutionary Computation [CD-ROM]. New York: ACM, 2011, p. 315-322. ISBN 978-1-4503-0557-0.

Kubalik, J., Faigl, J. (2006). *Iterative Prototype Optimisation with Evolved Improvement Steps*. In Proceedings of the 9th European Conferenceo Genetic Programming EuroGP 2006. Heidelberg: Springer, pp. 154-165. ISBN 3-540-33143-3.

Kubalik, J., Tichy, P., Sindelar, R., Staron, R.J. (2010). *Clustering Methods for Agent Distribution Optimization.* IEEE Transactions on Systems, Man, and Cybernetics: Part C. 2010, vol. 40, no. 1, p. 78-86. ISSN 1094-6977.

Kuramochi M., Karypis G. (2001). *Frequent Subgraph Discovery.* In ICDM, p. 313, First IEEE International Conference on Data Mining (ICDM'01).

Kuzelka O. and Zelezny F. (2008). *A Restarted Strategy for Efficient Subsumption Testing*. In Fundamenta Informaticae 89(1), pp. 95-109.

L. De Raedt. (2008). *Logical and Relational Learning*. Springer.

Lacroix V., Cottret L., Thibault P., Sagot M.-F. (2008). *An Introduction to Metabolic Networks and Their Structural Analysis*. In IEEE/ACM Trans. Comput. Biol. Bioinformatics 5, 4 (October 2008), pp. 594-617.

Landwehr N, Passerini A, DeRaedt L., and Frasconi P. (2010). *Fast learning of relational kernels*. In Mach. Learn. 78, 3 (March 2010), pp. 305-342.

Landwehr N., Kersting K., and De Raedt L. (2007). *Integrating Naïve Bayes and FOIL*. In J. Mach. Learn. Res. 8 (May 2007), pp. 481-507.

Landwehr, N., Kersting, K., and deRaedt, L. (2007). *Integrating Naive Bayes and FOIL*, in Journal of Machine Learning Research, 8, pp. 481–507.

Lavrac N. and Dzeroski S. (1994) *Inductive Logic Programming: Techniques and Applications*. Ellis Horwood.

Lavrac N., Flach P.A. and Zupan B. (1999). *Rule Evaluation Measures: A Unifying View.* In Proceedings of the 9th International Workshop on Inductive Logic Programming (ILP '99), Saso Dzeroski and Peter A. Flach (Eds.). Springer-Verlag, London, UK, pp. 174-185.

Ligtenberg W., Bosnacki D. and Hilbers P. (2009). *Mining Maximal Frequent Subgraphs in KEGG Reaction Networks*. In Proceedings of the 2009 20th International Workshop on Database and Expert Systems Application (DEXA '09). IEEE Computer Society, Washington, DC, USA, pp. 213-217.

Maloberti J. and Sebag M. (2004). *Fast theta-subsumption with constraint satisfaction algorithms.* In Machine Learning, 55(2):137-174.

Martin S., Mao Z., Chan L., and Rasheed S. (2007). *Inferring protein-protein interaction networks from protein complex data*. In Int. J. Bioinformatics Res. Appl. 3, 4 (October 2007), pp. 480-492.

Meinl T., Wörlein M., Urzova O., Fischer I. Philippsen M. (2006). *The ParMol Package for Frequent Subgraph Mining*, In Proceedings of the Third International Workshop on Graph Based Tools, ECEASST, ISSN 1863-2122.

Mellor D. (2008). *A learning classifier system approach to relational reinforcement learning*. In Jaume Bacardit, Ester Bernadó-Mansilla, Martin Butz, Tim Kovacs, Xavier Llorà, and Keiki Takadama, editors, *Learning Classifier Systems. 10th and 11th International Workshops (2006-2007)*, volume 4998/2008 of *Lecture Notes in Computer Science*, pages 169-188. Springer.

Mellor, D. (2008). *A Learning Classifier System Approach To Relational Reinforcement Learning*, Dissertation Thesis, University of Newcastle.

Metropolis, N., Rosenbluth, A. W., Rosenbluth, M. N., Teller, A. H., Teller, E. (1953). *Equation of State Calculations by Fast Computing Machines*. The Journal of Chemical Physics 21 (6): 1087.

Michalewicz, Z. (1992). *Genetic Algorithms + Data Structures = Evolution Programs*, NY: SpringerVerlag.

Mitchell T. (1997). *Machine Learning*, McGraw Hill

Mitchell M. (1996). *An Introduction to Genetic Algorithms*. MIT Press, Cambridge, MA

Michalski, R. S.(1980). Pattern Recognition as Rule-guided Inference. IEEE Transactions on Pattern Analysis and Machine Intelligence, PAMI, Vol.2, pp. 349–261.

Morishita S. (1998). *On classification and regression*, in Discovery Science, pp. 40–57.

Muggleton S. and Feng C. (1990). *Efficient Induction of Logic Programs.* In Proceedings of the 1st Conference on Algorithmic Learning Theory, pages 368–381. Japan.

Muggleton, S.H. (1991). *Inductive Logic Programming*, in New Generation Computing, 8(4), pp. 295-318.

Muggleton, S. (1995). *Inverse entailment and Progol*. New Generation Computing 13 (3–4): 245–286.

Muggleton S. and Raedt L. D. (1994). *Inductive Logic Programming: Theory and methods.* Journal of Logic Programming, 19/20:629–679.

Mühlenbein, H. (1992). *How Genetic Algorithms Really Work: Mutation and Hillclimbing*. In Proceedings of PPSN: pp. 15-26

Muchnick S. (1997). *Advanced Compiler Design and Implementation.* Morgan Kaufmann, USA

Neri F. and Saitta L. (1995). *A Formal Analysis of Selection Schemes*. Proc. Int. Conf. on Genetic Algorithms (Pittsburgh,PA), pp. 32-39.

Nijssen S. and Kok j.N. (2005). *The Gaston Tool for Frequent Subgraph Mining.* Electron. Notes Theor. Comput. Sci., 127 (1), pp. 77-87.

Nijssen S., Kok J. N. (2006). *Frequent Subgraph Miners: Runtimes Don't Say Everything*, In Proceedings of the International Workshop on Mining and Learning with Graphs (MLG 2006).

Noble C.C., Cook D.J. (2003). *Graph-based anomaly detection.* Proceedings of the 9th ACM SIGKDD international conference on Knowledge discovery and data mining, August 24-27, Washington, D.C. (2003)

O'Neill M. and Ryan C. (2003). *Grammatical Evolution: Evolutionary Automatic Programming in an Arbitrary Language*. Kluwer Academic Publishers, USA

Pei, J., Jiang, D., Zhang, A. (2005). *On Mining Cross-Graph Quasi-Cliques.* In: Proc. of KDD 2005

Quinlan J. R. (1990). *Learning logical definitions from relations*, Machine Learning, 5(3), pp. 239-266.

Quinlan. J. (1986). *Induction of Decision Trees*. In Machine Learning, 1(1), pp. 81–106.

R. Kohavi. (1995). *Wrappers for Performance Enhancement and Oblivious Decision Graphs*. PhD thesis, Stanford University.

Rechenberg I. (1988). *Artificial evolution and artificial intelligence*. In Machine Learning: Principles and techniques, R. Forsyth (Ed.). pp. 83-103. Chapman & Hall, Ltd., UK.

Reiser P. (1999). *Evolutionary Algorithms for Learning Formulae in First-order Logic*, Dissertation thesis, University of Wales.

Reiser P. and Riddle P. (1998). *Evolving Logic Programs to Classify Chess-Endgame Positions.* In Selected papers from the Second Asia-Pacific Conference on Simulated Evolution and Learning on Simulated Evolution and Learning (SEAL'98), pp. 138-145, Springer-Verlag, London.

Reiser P. and Riddle P. (2001). *Scaling Up Inductive Logic Programming: An Evolutionary Wrapper Approach*. In Applied Intelligence 15, 3 (July 2001), pp. 181-197.

Reiter R. (1978). *On closed world data bases*. In H. Gallaire and J. Minker, editors. Logic and Data Bases. Plenum Press, pp. 55-76, New York.

Richard A. M.. (1999). *Application of artificial intelligence and computer-based methods to predicting chemical toxicity*. In Knowl. Eng. Rev. 14, 4 (December 1999), pp. 307-317.

Rissanen J. (1978). *Modeling by shortest data description* In Automatica, 14, pp. 465-471

Rückert U. and Kramer S. (2002). *Stochastic Local Search In k-Term DNF Learning*, in Proc. of 20th Intl. Conf. on Machine Learning, pp. 648-655.

Sanchez, E., Squillero, G., Tonda, A. (eds.). (2012). *Industrial Applications of Evolutionary Algorithms*, Springer

Sebag M. and Rouveirol C. (1996). *Constraint Inductive Logic Programming.* In L. De Raedt (ed.). Advances in Inductive Logic Programming. IOS Press, pp. 277–294.

Sebag M. and Rouveirol C. (2000). *Resource-bounded relational reasoning: induction and deduction through stochastic matching*. In Machine Learning, 38, pp. 41-62.

Selman B., Levesque H.J., Mitchell D. (1992). *A new method for solving hard satisfiability problems*, In P. Rosenbloom, P. Szolovits (Eds.), Proceedings of the Tenth National Conference on Artificial Intelligence, pp. 440–446, AAAI Press.

Serrurier M., Prade H., and Richard G. (2004) *A Simulated Annealing Framework for ILP*, in R. Camacho, R. King, A. Srinivasan (Eds.): ILP 2004, LNAI 3194, pp. 288–304, Springer-Verlag Berlin Heidelberg.

Schwefel H.-P. (1977). *Numerische Optimierung von Computer-Modellen mittels der Evolutionsstrategie*, In Interdisciplinary systems research; 26. Birkhäuser, Basel

Slattery S. and Craven M. (1998). *Combining statistical and relational methods for learning in hypertext domains*. In Proceedings of the 8th International Conference on Inductive Logic Programming (ILP-98), 1998.

Sokolova M., Japkowicz N., and Szpakowicz S. (2006). *Beyond Accuracy, F-Score and ROC: A Family of Discriminant Measures for Performance Evaluation*, In AI 2006: Advances in Artificial Intelligence, pp. 1015 – 1021, Springer.

Srinivasan A., Camacho R. (1993) *The Aleph Manual*, available at http://www.comlab.ox.ac.uk/oucl/research/areas/machlearn/Aleph, last checked on Dec 1st, 2012

Srinivasan, A., King, R. D., & Bristol, D. W. (1999). *An Assessment of ILP-Assisted Models for Toxicology and the PTE-3 Experiment.* In Proceedings of the 9th International Workshop on Inductive Logic Programming, pp. 291-302.

Srinivasan A. (1999). *A study of two sampling methods for analysing large datasets with ILP*. In Data Mining and Knowledge Discovery, 3, pp. 95-123.

Srinivasan A. (2001). *Extracting Context-Sensitive Models in Inductive Logic Programming*. In Mach. Learn. 44, 3 (September 2001), pp. 301-324.

Stetter M., Deco G., and Dejori M. (2003). *Large-Scale Computational Modeling of Genetic Regulatory Networks*. Im Artif. Intell. Rev. 20, 1-2 (October 2003), 75-93.

Struyf J. and Blockeel H. (2003). *Query Optimization in Inductive Logic Programming by Reordering Literals*, In Inductive Logic Programming, Lecture Notes in Computer Science, Volume 2835/2003, pp. 329-346, Springer-Verlag.

Tamaddoni-Nezhad A. and Muggleton S. (2002) *A Genetic Algorithms Approach to ILP*, in Proceedings of ILP 2002, 12th Intl. Conference on Inductive Logic Programming, Sydney, Australia.

Turcotte, M., Muggleton, S.H., Sternberg, M.J.E. (2001). *Automated Discovery of Structural Signatures of Protein Fold and Function*. In J. Mol. Biol. 306, pp. 591-605.

Ullmann, J. R. (1976). *An Algorithm for Subgraph Isomorphism*. Journal of the ACM, 23(1), pp.31–42

Valiant L. G. (1984). *A theory of the learnable*, Communications of the ACM 27(11), pp. 1134-1142

Van Laer W. and De Raedt L. (2000). *How to Upgrade Propositional Learners to First Order Logic: a Case Study*, In Relational Data Mining, 235–256, Springer-Verlag New York, Inc., New York, NY, USA,

Vanetik N., Gudes E. and Shimony S. E. (2002). *Computing frequent graph patterns from semistructured data*, In Proceedings of 2002 IEEE International Conference on Data Mining (ICDM'02). Maebashi City, Japan; pp. 458, ISBN:0-7695-1754-4 IEEE Computer Society.

Washio T., Motoda H. (2003). *State of the Art of Graph-based Data Mining*, In SIGKDD Explorations Special Issue on Multi-Relational Data Mining, 5 (1), pp 59-68.

Watts D.J. (1999). *Small Worlds: The Dynamics of Networks Between Order and Randomness*, Princeton University Press, Princeton.

Wong M. L. and Leung K. S. (1995). *Inducing Logic Programs With Genetic Algorithms: The Genetic Logic Programming System*, in IEEE Expert 10(5), pp. 68-76.

Wrobel S. (1997). *An algorithm for multi-relational discovery of subgroups*. In Proceedings of the First European Symposium on Principles of Data Mining and Knowledge Discovery, pp. 78–87. Springer-Verlag.

Yan X., Han J. (2002). *gSpan: Graph-Based Substructure Pattern Mining*, Proceedings of the 2002 IEEE International Conference on Data Mining (ICDM 2002), IEEE Computer Society, pp. 721-724.

Zelezny F., Lavrac N. (2006). *Propositionalization-Based Relational Subgroup Discovery with RSD.* In Machine Learning 62(1-2), pp. 33-63

Zhang L. and Cui Y. (2010). *An efficient method for DNA-based species assignment via gene tree and species tree reconciliation*. In Proceedings of the 10th international conference on Algorithms in bioinformatics. Springer-Verlag, Berlin, Heidelberg, pp. 300-311.

Zucker, J.-D., & Ganascia, J.-G. (1998). *Learning structurally indeterminate clauses*. In Proc. of the 8th International Conference on ILP. Springer-Verlag.

Žáková, M., Železný, F., Garcia-Sedano, J., Tissot, C. M., Lavrač, N., Křemen, P. and Molina, J. (2007). *Relational Data Mining Applied To Virtual Engineering Of Product Designs*. International Conference on Inductive Logic Programming (ILP '07). Springer.

Železný F., Srinivasan A., Page D. (2006). *Randomized Restarted Search in ILP*, in Machine Learning 64(1—2): pp.183-208.