Mach Learn (2017) 106:197–241
DOI 10.1007/s10994-016-5595-3

On the use of stochastic local search techniques to revise first-order logic theories from examples

Aline Paes[1] · **Gerson Zaverucha**[2] · **Vítor Santos Costa**[3]

Received: 19 May 2012 / Accepted: 18 October 2016 / Published online: 15 December 2016

Abstract Theory Revision from Examples is the process of repairing incorrect theories and/or improving incomplete theories from a set of examples. This process usually results in more accurate and comprehensible theories than purely inductive learning. However, so far, progress on the use of theory revision techniques has been limited by the large search space they yield. In this article, we argue that it is possible to reduce the search space of a theory revision system by introducing stochastic local search. More precisely, we introduce a number of stochastic local search components at the key steps of the revision process, and implement them on a state-of-the-art revision system that makes use of the most specific clause to constrain the search space. We show that with the use of these SLS techniques it is possible for the revision system to be executed in a feasible time, while still improving the initial theory and in a number of cases even reaching better accuracies than the deterministic revision process. Moreover, in some cases the revision process can be faster and still achieve better accuracies than an ILP system learning from an empty initial hypothesis or assuming an initial theory to be correct.

Keywords Inductive logic programming · Theory revision from examples · Stochastic local search

Editor: Kristian Kersting.

✉ Aline Paes
alinepaes@ic.uff.br

Gerson Zaverucha
gerson@cos.ufrj.br

Vítor Santos Costa
vsc@dcc.fc.up.pt

1 Department of Computer Science, Universidade Federal Fluminense (UFF), Niterói, RJ, Brazil

2 Department of Systems Engineering and Computer Science - COPPE, Universidade Federal do Rio de Janeiro (UFRJ), Rio de Janeiro, RJ, Brazil

3 CRACS and DCC/FCUP, Universidade do Porto, Porto, Portugal

1 Introduction

Inductive Logic Programming is the process of automatically learning First-Order Logic Theories from a set of examples and a fixed body of prior knowledge, the *background knowledge*. A large number of algorithms and systems have been developed towards this goal. Popular examples include FOIL (Quinlan 1990), Claudien (De Raedt and Bruynooghe 1993), Progol (Muggleton 1995), Tilde (Blockeel and De Raedt 1998), Aleph (Srinivasan 2001) and ProGolem (Muggleton et al. 2010b), among many others. Most such systems start from an empty initial hypothesis. Thus, we say that they *learn from scratch*. However, it may be the case that an incomplete or only partially correct theory exists. Such a theory may have been elicited by a domain expert who relies on incorrect assumptions or who only has partial, but still useful, understanding of the domain. Or maybe new examples, that cannot be explained by the current theory, have become available. Or we may simply want to improve a theory learned from scratch. In all such cases, since the initial theory probably contains important information, one would like to use it as a starting point for the learning process, modify it and ultimately improve it. Ideally, after the modifications the theories should be more accurate.

These considerations have motivated the development of several *theory refinement* systems (Shapiro 1981; Buntine 1991; Wogulis and Pazzani 1993; Towell and Shavlik 1994; Adé et al. 1994; Wrobel 1994; Richards and Mooney 1995; Wrobel 1996; Ramachandran and Mooney 1998; Garcez and Zaverucha 1999; Esposito et al. 2000). Such systems assume that the initial theory is approximately correct. If so, then only some *points* (clauses and/or literals) in the theory prevent it from correctly modeling the dataset. Therefore, it should be more effective to search for such points in the theory and to *revise* them than to either discard the initial theory or to propose modifications to all its clauses.

Usually, a theory revision system performs search in three main steps:

1. They search for clauses and literals in the theory responsible for the misclassification of some example.
2. They yield possible modifications to such points through applying at each point a number of matching revision *operators*. Commonly used revision operators are *deletion* from and *addition* of literals to the body of existing clauses and deletion and addition of rules.
3. They score the proposed revisions and select the best.

The size of the search space depends on the number of misclassified examples, on the number of clauses and literals in the theory responsible for misclassified examples and also on the size of the knowledge base, which is used to yield and score possible modifications. Moreover, theory revision systems tackle whole theories instead of performing stepwise search for individual clauses as most ILP systems do. Search over whole theories is known to be a hard problem (Bratko 1999). Indeed, traditional theory revision systems must search over extremely large spaces, and can become rather inefficient. On the other hand, they usually produce more accurate and comprehensible theories than purely inductive learning. As Dietterich et al. (2008) argued, current revision systems explore large search spaces, however, the fast development of rich knowledge bases in areas such as biology (Muggleton 2005) suggests that there may be a need for theory revision.

A first step to reduce the cost of the revision process is to replace purely top-down literal generation of the revision system FORTE (Richards and Mooney 1995) to use the more efficient hybrid bottom-up and top-down approach to refine clauses (Duboc et al. 2009) by a *Mode Directed Inverse Entailment (MDIE)* approach (Muggleton 1995). However, when

facing large initial theories and background knowledge this is not enough to make the revision process efficient. Ultimately, revision systems must search over large spaces.

The last years have shown that stochastic local methods, originally designed to solve difficult combinatorial propositional problems (Selman et al. 1992, 1996), can also perform well in a variety of applications (Chisholm and Tadepalli 2002; Rückert and Kramer 2003, 2004). This led to interest in applying such techniques on data-mining applications, and more specifically on multi-relational data-mining. Initial work on the area has indeed shown very substantial improvements in efficiency, with little or no cost in accuracy, when learning theories from scratch in ILP systems (Srinivasan 2000; Železný et al. 2002, 2006; Paes et al. 2006; Muggleton and Tamaddoni-Nezhad 2008).

Initial experiments with the theory revision system FORTE showed good promise from introducing stochastic search when searching for the literals to be added/deleted to/from a clause and when searching the revision to be implemented (Paes et al. 2007). This paper enhances that work by including a stochastic search at each step of the revision process. The system presented in this paper is called *YAVFORTE* and, besides the stochastic algorithms, it makes use of the most specific clause as in Duboc et al. (2009), and it includes improvements in the top-level algorithm and in both the deletion and the addition of antecedents processes within the revision operators. The experimental results presented in this paper show that the running time of the revision process is reduced by the use of SLS techniques, even being competitive with a standard ILP system, while the accuracies are higher than the ones obtained by the standard inductive system. This last benefit is also observed in previous work (Richards and Mooney 1995; Duboc et al. 2009).

The outline of the paper is as follows. Firstly, we review main Stochastic Local Search techniques in Sect. 2. Next, we review theory revision and present the main modifications implemented on the FORTE system that originates YAVFORTE in Sect. 3. The algorithms developed to revise FOL theories from examples through SLS are devised in Sect. 4. Experimental results are presented in Sect. 5, followed by conclusions and future work in Sect. 6.

2 Stochastic local search

In order to get good hypotheses while still keeping the search feasible, one may take advantage of local search algorithms, which start by generating a candidate hypothesis at some location in the search space and afterward move from the present location to a neighboring location in the search space. If one sees the search space as a graph, the neighbor of a node N in the graph space is any other node M that is directly connected to N. Each location has a relatively small numbers of neighbors and each move is determined by a decision based on *local* knowledge (Hoos and Stützle 2005). In this way, they abandon completeness to gain efficiency.

To further improve efficiency and also escape from local optima, one may use randomized choices when generating or selecting candidates in the search space of a problem, through *Stochastic Local Search Algorithms* (SLS). One major motivation and successful application of SLS has been in satisfiability checking of propositional formulae, namely through the well-known GSAT (Selman et al. 1992) and WalkSAT (Selman et al. 1996) algorithms. A large number of tasks in areas such as planning, scheduling and constraint solving can be encoded as a satisfiability problem, and empirical observations show that SLS often can substantially improve their efficiency (Chisholm and Tadepalli 2002; Rückert and Kramer 2003).

2.1 Stochastic local search methods

The key ideas of the search process performed by a Stochastic Local Search Algorithm are as follows.

1. **Initialization** An initial candidate solution is selected, usually by generating a candidate at random;
2. **Move** Iteratively, the process (at random) decides to move from the present candidate solution to a local neighboring candidate solution, usually (but not always) considering a function to evaluate the neighbors.
3. **Stopping** The process is finished when it attends a termination criteria, which could be a maximum number of iterations or a solution has been found.

There are several different strategies to follow when implementing a Stochastic Local Search technique. Next we briefly describe some of the families of SLS strategies, which are used in this work.

Stochastic Hill Climbing (Russell and Norvig 2010)—this strategy executes in three steps:

1. **Initialization** start from a randomly selected point in the search space.
2. **Move** choose with uniform probability distribution a neighbor of the current candidate, requiring the value of the evaluation function to improve.
3. **Stopping criteria** finish when none of the neighbors improves the evaluation function (stopping criteria).

Randomized Iterative Improvement (RII) (Hoos and Stützle 2005)—in this strategy, the search alternates with a fixed frequency between selecting an improving neighbor and selecting a neighbor at random. A Randomized Iterative Improvement algorithm does not terminate as soon as a local optimum is encountered. Instead, it may stop the execution when it reaches a number of iterations or when a number of search steps have been performed without making progress.

The most famous algorithm in this family is arguably WalkSAT (Selman et al. 1996). WalkSAT was developed to check satisfiability of propositional formulae. WalkSAT decides at each step with a fixed probability np whether to do a standard greedy step or to flip a variable selected uniformly at random from the set of all variables occurring in unsatisfied clauses. The probability np is called *walk probability, noise setting or noise level*. WalkSAT starts from a randomly generated assignment of the variables in an initial formula and considers a maximum number of tries and a maximum number of steps in order to find the solution.

Following the success of GSAT/WalkSAT based algorithms, a large number of randomized strategies were derived from them to solve tasks in areas such as planning, scheduling, constraint solving and rule learning (Chisholm and Tadepalli 2002; Rückert and Kramer 2003).

Probabilistic Iterative Improvement (PII) —in each step of the search process a PII algorithm selects a neighbor according to a given function $p(g, s)$, which determines a probability distribution over neighboring candidate solutions of s based on their evaluation function value g. A bad neighbor candidate can be accepted depending on the deterioration in the evaluation function value. In other words, the worse a step is, the less likely it is to be performed.

Stochastic local search algorithms have been successfully applied to machine learning algorithms, both in propositional learning and in the relational setting (Paes et al. 2006;

Železný et al. 2002, 2006; Chisholm and Tadepalli 2002; Rückert and Kramer 2003; Tamaddoni-Nezhad and Muggleton 2000; Muggleton and Tamaddoni-Nezhad 2008; Serrurier and Prade 2008; Joshi et al. 2008; Specia et al. 2009).

3 Theory revision from examples and YAVFORTE

ILP algorithms learn first-order clauses given a set of examples and a static and assumed as correct background knowledge (BK). On the other hand, *theory revision from examples* (Wrobel 1996) has the goal to improve a previously obtained knowledge. To do so, theory revision assumes the BK may also contain incorrect rules, which should be modified to better reflect the set of examples. Revision in certain clauses of the BK can be avoided by letting a part of the preliminary knowledge defined as correct and invariant. Thus, in theory revision the BK is divided into two parts: a set of rules assumed as correct and therefore not modifiable, called here as *Fundamental Domain Theory (FDT)* (Richards and Mooney 1995); and the remaining rules which may be incorrect and are subject to modifications, called the *Initial Theory*. The goal of a theory revision process is to identify points in the initial theory which prevent it from correctly classifying positive or negative examples, and propose modifications to such points, so that the revised theory together with the FDT is as close to a correct theory as possible. The task of theory revision from examples is defined as follows (Wrobel 1996).

Definition 1 Given:

- A background knowledge BK written as definite clauses, divided into
  - A modifiable set of clauses which might be incorrect (H') and
  - An invariant and assumed as correct set of clauses (FDT) and
- A set of positive E^+ and negative examples E^- composing the set of examples E, written as ground definite clauses

Find:

- A revised theory H consisting of definite first-order clauses such that
  - itemize$FDT \wedge H \models E^+$ (H is complete) and $FDT \wedge H \not\models E^-$ (H is consistent), i.e., H is correct.
  - H obeys a minimality criteria such that it is as syntactically and semantically close as possible to the original BK.

Usually, it is not possible to find a correct theory, i.e., a complete and consistent theory. Thus, the correctness criteria is relaxed to find a theory as close as possible to be correct.

The task of an ILP system, on the other hand, is defined as

Definition 2 Given:

- An invariant and assumed as correct background knowledge BK written as definite clauses
- A set of positive E^+ and negative examples E^- composing the set of examples E, written as ground definite clauses

Find:

- A set of clauses H consisting of definite first-order clauses such that $BK \wedge H \models E^+$ (H is complete) and $BK \wedge H \not\models E^-$ (H is consistent), i.e., H is correct .

Note that, while in the theory revision task we have an initial hypothesis H' which is normally nonempty and will be modified during the revision process, in ILP there is no such an initial hypothesis, as the BK is fixed. Thus, we say in this paper that ILP systems learn from scratch, since the initial hypothesis is empty.

Theory Revision is particularly powerful and challenging because it must deal with the issues arising from revising multiple clauses (theory) and even multiple predicates (multiple target concepts). Additionally, as the initial theory is a good starting point and the revision process takes advantage of it, the theories returned by revision systems are usually more accurate than theories learned from standard ILP systems using the same dataset. Several papers such as Shapiro (1981), Wogulis and Pazzani (1993), Richards and Mooney (1995), Buntine (1991), Towell and Shavlik (1994), Adé et al. (1994), Wrobel (1996), Ramachandran and Mooney (1998), Garcez and Zaverucha (1999), Esposito et al. (2000) show that propositional and first-order theory revision systems are capable of learning more compact and accurate theories than purely inductive systems even using less examples.

Revision Points Revision systems work by identifying and trying to solve misclassified examples. A positive example not covered by the theory (a *false negative*) indicates the theory is too specific and, therefore, needs to be *generalized*. In the opposite case, a negative example covered by the theory (a *false positive*) indicates it is too general and therefore it needs to be *specialized*.

Often, many clauses can be involved in proving negative examples; moreover, many clauses could be generalized so that the misclassified positive examples would be covered. In theory revision, all such clauses and literals are called *revision points*, defined as follows.

Definition 3 Let H' be a theory that can be modified and FDT a set of clauses assumed as correct. Let P be a path in a SLD tree (Kowalski and Kuehner 1971; Lloyd 1987), whose root is an example $e \in E$. A node $n \in P$ is an objective clause and an edge $edge(n_1, n_2) \in P$ is composed of a θ-substitution and a clause $\in H' \bigcup FDT$, whose head is the first literal in n_1. P is a *refutation path* if its leaf is an empty clause $\square$ or a *failure path* otherwise.

Definition 4 Let P^+ be a refutation path in a SLD tree whose root is a negative example $e^- \in E^-$. Let the clauses from each edge in P^+ be $\{C_1, C_2., \ldots, C_n\}$. Each clause $C_1, C_2., \ldots, C_n \cap H'$ is a *specialization revision point*.

Definition 5 Let P^- be a failure path in a SLD tree whose root is a positive example $e^+ \in E^+$ and whose leaf has the literal $pred(T_1, \ldots, T_m)$ in the objective clause. Then,

- the variabilized version of $pred(T_1, \ldots, T_m)$ is a generalization revision point;
- any literal appearing in the path before $pred(T_1, \ldots, T_m)$ that has bounded at least one variable in $\{(T_1, \ldots, T_m\}$ is a generalization revision point;
- the clauses that have the literals above in their body are generalization revision points.

To sum up, *specialization revision points* are clauses in the theory used in successful proof paths of negative examples. Arguably, modifications on such clauses will make such misclassified negative examples to become unprovable. On the other hand, *generalization revision points* are literals and clauses in the theory responsible for or contributing to the failure of positive examples. Arguably, by changing such literals and/or clauses, misclassified positive examples can become provable.

Revision Operators In order to modify the theory in the selected revision points, theory revision systems rely on *revision operators* that propose modifications at each revision point. The

type of the revision point determines the revision operator that will be applied to try to make the theory consistent with the dataset. One may consider two types of revision operators: *generalization* operators, applied on generalization revision points and *specialization* operators, applied on specialization revision points (Wrobel 1996). Any operator used in first-order machine learning can be used in a theory revision system. Specialization operators decrease coverage and can be used to remove false positives:

- **Delete-rule** this commonly used operator removes a clause that was used to prove a negative example.
- **Add-antecedents** this operator adds antecedents to an inconsistent clause, that has been previously marked as a revision point.

Generalization operators address false negatives. Two commonly used generalization operators are:

- **Delete-antecedents** this operator removes failed antecedents marked as revision points from clauses that could be used to prove positive examples.
- **Add-rule** this operator generates new clauses, either from failed existing clauses (deleting antecedents followed by addition of antecedents) or from scratch (starting only from the generalized head of the example).

In addition, FORTE (Richards and Mooney 1995) has two generalization revision operators based on inverse resolution (Muggleton 1995), namely:

- **Identification** this operator performs an inverse resolution step in two rules of the theory, to provide an additional rule for a faulty predicate.
- **Absorption** different from the previous operator, this one does not create a new clause for a faulty predicate, but instead tries to find an existing clause whose antecedents subsume the failing literal and which has alternate clauses to allow the failing positive instances to be proven.

These six operators are the ones used in this work. For more revision operators, we refer the reader to Wrobel (1996).

3.1 YAVFORTE

Although theory revision systems usually induce more accurate theories than standard ILP techniques, the revision process is expensive, mainly because theory revision refines whole theories instead of individual clauses (Wrobel 1996; Bratko 1999).

Mode Directed Inverse Entailment (MDIE) (Muggleton 1995) is often used to bound the search space of new literals. We recently showed that the runtime of FORTE (Richards and Mooney 1995)—a standard revision system—can be greatly reduced by using MDIE (Duboc et al. 2009). The algorithm for specializing clauses from the literals of the Bottom Clause during the revision process is named FORTE_MBC.

YAVFORTE *(Yet Another Version of FORTE)*, improves FORTE_MBC further. Next, we describe the top-level algorithm.

YAVFORTE Top Level Algorithm YAVFORTE is built upon the FORTE Algorithm. As the original, it performs a hill climbing search through a space of specialization and generalization operators. However, YAVFORTE always starts from the less costly operator, and immediately stops when finding an operator that achieves the maximum score. The score is the output of an evaluation function computed over the proposed revision and the examples. For instance,

the score of a proposed revision could be the number of incorrect examples that have become correctly proven after proposing the revision, minus the number of correct examples that have become proven incorrectly. Moreover, YAVFORTE, allows the user to specify which revision operators are to be employed. One can restrict oneself to only apply specialization or only generalization or to a subset of both generalization and specialization operators.

Algorithm 1 presents the revision process of YAVFORTE. Within a single iteration, the algorithm finds the revision points and sorts them according to their potential. The potential of a revision point is defined as the number of examples that has pointed out the necessity of revising it. Next, for each revision point, matching revision operators are proposed, until a maximum score or a maximum potential is reached by one revision operator. The revision with the best score is chosen and implemented in case the overall score is indeed improved. The process continues until no revision is able to improve the current score.

Algorithm 1 YAVFORTE Top-Level Algorithm

Input: An initial theory T, background knowledge FDT, a set of examples E, list of applicable operators Rev
Output: A revised theory T'

```
if Rev = ∅ then
    Rev ← all revision operators
GenRev ← ordered list of generalization operators in Rev
SpecRev ← ordered list of specialization operators in Rev
repeat
    generate revision points;
    sort revision points by potential (high to low);
    for each revision point RP do
        if RP is a specialization revision point then
            for each revision operator RO ∈ SpecRev do
                Generate all applications of RO in RP
                compute score_RO
        else
            for each revision operator RO ∈ GenRev do
                Generate all applications of RO in RP
                compute score_RO
        update best revision found;
        until score_RO = maximum score or RO achieved maximum potential of RP
    if best revision improves the theory then
        implement best revision;
until no revision improves the theory;
```

3.2 Finding revision points

The algorithms for finding revision points in the deterministic component of YAVFORTE are identical to the ones devised in FORTE system. They identify revision points by annotating proofs of incorrectly provable negative instances or by annotating attempted proofs of incorrectly unprovable positive instances. When the goal is to find specialization revision points, all the provable instances are considered, since they are the ones whose provability may be affected by a specialization in the theory: any of these instances might become unprovable because of the specialization. These instances are either true positive-correctly classified positive instances—or false positives-misclassified negative instances. The algorithm for collecting specialization revision points is shown in Algorithm 2.

Algorithm 2 FORTE Algorithm for collecting specialization revision points

Input: The current theory H'; the fixed preliminary knowledge FDT; the set of provable intances $EP = TP \cup FP$

Output: RPS, a set of clauses marked as specialization revision points, each one annotated with TPC, true positive instances relative to clause C, FPC, false positive instances relative to clause C, and PC the potential of the revision point

```
1: for each provable instance e ∈ EP do
2:    Ce ← clauses participating in the proof of the instance e using H' and FDT
3:    for each clause C ∈ Ce do
4:       if e ∈ FP then
5:          FPC ← FPC ∪ e;
6:          PC ← PC + 1;
7:       else
8:          TPC ← TPC ∪ e;
9:    RPS ← RPS ∪ Ce;
10: for each clause C ∈ RPS do
11:    if PC = 0 then
12:       delete C from RPS;
```

First, the algorithm annotates each clause participating in the successful proof of the instances. The positive instances are annotated separately from the negative instances in the clauses. In case the clause has no annotation of false positive instances, it is discarded. The remaining clauses compose the set of specialization revision points. The true positive and false positive instances relative to the clauses are used to compute the potential of the revision points and later to calculate the score of the revision proposed to those points.

In case there are misclassified positive instances, the goal is to find generalization revision points. In this case, all the unprovable instances are considered, since they are the ones whose provability may be affected by a generalization in the theory: any of these instances might become provable because of the generalization. These instances are either true negatives-correctly classified negative instances—or false negatives-misclassified positive instances. In order to identify generalization revision points, it is necessary to make annotations from failed proofs of positive instances. Thus, each time a backtrack occurs, the failed literal is noted and marked as a failure point. Next, the literals binding values to variables in failure points are collected recursively and also marked as failure points. Finally, the clauses with the failure literals are also marked as failure points. This process is also followed to identify which failure points are responsible for not proving negative instances, since they might become provable after a revision in such points. The list of TN and FN instances are used to calculate the potential of the revision point, and, after proposing some revision, to calculate the score of the revision on such a point. The procedure is exhibited as Algorithm 3.

3.3 YAVFORTE revision operators

As stated before, FORTE has two specialization operators, namely, delete-rule and add-antecedents. Delete-rule works by simply proposing the removal of a clause participating on the proof of negative examples. This operator is not modified regarding the original. Add-antecedents is modified to consider the Bottom Clause to bound the space of new literals (Duboc et al. 2009). We change the implementation of FORTE_MBC so that each term in the clause under specialization matches a type and each literal matches a mode definition, *before* constructing the Bottom clause. In this way, the terms of the current clause are considered when saturating a positive instance. Following FORTE, YAVFORTE is able to add

Algorithm 3 FORTE Algorithm for collecting generalization revision points

Input: The current theory H'; the fixed preliminary knowledge FDT; the set of unprovable instances $EU = TN \cup FN$

Output: RPG, a set of clauses marked as generalization revision points, each one annotated with TNC, true negative instances relative to clause C, FNC, false negative instances relative to clause C, and PC the potential of the revision point

for each unprovable instance $e \in EU$ **do**
  Try to prove instance e using H' and FDT
  for each time that a literal fails **do**
    collect the failed literal lfe;
    collect the literals LCe responsible for binding variables in lfe, recursively
    collect the clauses Ce where lfe failed and where LCe appeared
    if $e \in TN$ **then**
      $TN\_lfe \leftarrow TN\_lfe \cup e$
      for each literal $lce \in LCe$ **do**
        $TN\_lce \leftarrow TN\_lce \cup e$
        for each clause $ce \in Ce$ **do**
          $TN\_ce \leftarrow TN\_ce \cup e$
    else
      $FN\_lfe \leftarrow FN\_lfe \cup e$
      $P\_lfe \leftarrow P\_lfe + 1$
      for each literal $lce \in LCe$ **do**
        $FN\_lce \leftarrow FN\_lce \cup e$
        $P\_lce \leftarrow P\_lce + 1$
        for each clause $ce \in Ce$ **do**
          $FN\_ce \leftarrow FN\_ce \cup e$
          $P\_ce \leftarrow P\_ce + 1$
    $RPG \leftarrow RPG \cup lfe \cup LCe \cup Ce$
for each point $P \in RPG$ **do**
  if $PG = 0$ **then**
    delete P from RPG;

antecedents to a clause using a standard hill climbing approach and the relational pathfinding algorithm (Richards and Mooney 1992), whose candidate literals are also collected from the Bottom clause.

Concerning generalization, FORTE has four different operators, namely, delete-antecedents, add-rule, absorption, and identification. YAVFORTE lets absorption and identification intact. Add-rules operator produces new rules either by adding literals to the body of a generalized example or by deleting antecedents from existing clauses and adding antecedents to this generalized clause. This additional second step is necessary to avoid proof of negative examples that become provable exactly because of the antecedents deleted. In both cases, adding rules to the theory requires that new literals be generated, to compose the set of candidate antecedents. Thus, the Bottom Clause is also used within this operator to bound the search space of candidate literals.

The process of deleting antecedents in YAVFORTE differs from FORTE (and FORTE_MBC) in two aspects: (1) YAVFORTE only deletes antecedents that produce final clauses obeying the modes language and (2) the narrow requirement of only deleting antecedents when none of the negative examples become provable is no longer sought, i.e., YAVFORTE allows noisy data by relaxing the criteria of the delete antecedents operator.

3.3.1 Adding antecedents to clauses in YAVFORTE

FORTE, following FOIL (Quinlan 1990), generates literals to be added to a clause obeying two conditions: (1) the variables of the literals must follow their types defined in the knowledge base and (2) they must have at least one variable in common with the current clause. While this makes the generation of antecedents simple and fast, it also leads to a large search space of candidate antecedents, composed of all the possible literals of the knowledge base. Such a large search space turns the complexity of the Add-antecedent operator very high, contributing to the bottleneck of the revision process. Aiming at reducing such cost, we implemented the following modifications in FORTE_MBC:

1. The variabilized Bottom Clause is the search space of literals, which reduces the number of candidate literals and also imposes the following constraints:
   - Limits the maximum number of different instantiations of a literal (the recall number);
   - Limits the number of new variables in a clause;
   - Guarantees that at least one positive example is covered (the one which generates the Bottom Clause).
2. The mode declarations are used to further constrain the antecedents, which means they will only be added to the clause if their use respects the mode defined in the knowledge base.
3. Determination definitions of the form

$$determination(HeadPredicate/Arity, BodyPredicate/Arity)$$

   state which predicates can be called in the clauses defining $HeadPredicate$.

The Bottom Clause is created immediately before the search for antecedents begins, by saturating a positive example covered by the clause being specialized (the base clause). The Bottom Clause is going to be composed of the literals relevant to at least such a positive example and it is guaranteed to be a super-set of the base clause.

The process of constructing a Bottom Clause in YAVFORTE differs from the classical Bottom Clause construction Algorithm when the base clause has a non-empty body, since in this case it is necessary to take into account the terms of the current clause. Note that in Duboc et al. (2009) the variables of the Bottom Clause were unified with the variables of the base clause only *after* the Bottom Clause had been constructed. Besides this being an expensive process involving lots of backtracks, sometimes it was not possible to find a correct unification. We noticed two of such situations: in cases where the example contains two or more equal terms in the head, but the base clause has two different variables in their place (1) and when the base clause has no constant in the head but the first `modeh` has a constant, or vice-verse (2). In these cases, the Bottom Clause would follow the unification according to the example. The result would be a Bottom Clause with variables different from the ones in the base clause which would make more difficult to find a proper substitution matching both the Bottom Clause and the base clause.

Thus, the first step when constructing the Bottom clause is to find the ground literals of the base clause considering the example but maintaining the substitution of the variables in the base clause. Then, the terms of the base clause are put in the list of terms to be used by the procedure and the Bottom Clause is initialized with the literals of the base clause. The rest of the process is the same as the original algorithm, except that we do not allow the inclusion of a literal being already in the Bottom Clause. Note that if the clause is being constructed

from scratch, it is not necessary to keep an association with the base clause passed as an input argument. In this case, we completely follow the original algorithm.

An important difference between the algorithm developed in Duboc et al. (2009) and the one used in this work concerns intermediate clauses. These are clauses that do not have directly associated examples, i.e., there is no example with the same predicate as the one in the head of such clauses, and the predicate in the head of such clauses may appear in the body of others clauses. Such clauses may also need to be specialized, by adding literals to them, taken from a Bottom clause. Nevertheless, the Bottom clause is constructed from a positive example covered by the clause, whose example predicate is the same as the one in the head of the clause. The problem here is that not necessarily we have examples of such intermediate predicates.

Therefore, we need to tackle non-observation predicate learning (Muggleton and Bryant 2000), where the concept being learned differs from that observed in the examples. We introduced in a previous work *intermediate predicate abduction* in order to "fabricate" the required example (Muggleton et al. 2010a). From a positive instance belonging to the relevant examples of the intermediate clause, i.e, the proof of the instance includes the clause, we obtain an "intermediate instance" using the current theory and the FDT. The procedure instantiates such a predicate to its first call encountered when attempting to prove the goal. The proof starts with the example and finds an instantiation for the specified intermediate predicate. After constructing the intermediate example, the bottom clause construction procedure is ready to run, followed by the refinement of the clause. The whole procedure can be visualized in Algorithm 4.

Note that the original FORTE system does not have this problem, as it builds the search space of candidate antecedents following a top-down approach that does not require examples to work.

4 Stochastic local search to revise first-order logic theories from examples

Theory revision is known to be an expensive task. Algorithm 1 shows the three main steps that contributes to an increasingly running time:

1. Generating revision points (line 6).
2. Generating revisions (lines 10 and 14).
3. Proposing modifications to clauses, through addition and deletion of antecedents (lines 11, 12 and 15, 16).

In order to improve efficiency, we propose to apply stochastic components to each key step above:

1. **Search for revision points** a random decision may return a subset of the revision points instead of always returning all of them.
2. **Revision search** rather than proposing all revisions, one might enumerate the possible modifications and choose one to implement at random.
3. **Antecedents search** as proposals of modifications are dominated by the addition and deletion of antecedents, one may benefit from randomizing antecedents search.

Preliminary experiments with the theory revision system FORTE showed good promise from introducing stochastic search at the last two searches above (Paes et al. 2007). This paper enhances that work by including a number of stochastic components at every step of the YAVFORTE system. While in the former work the stochastic components were introduced

Algorithm 4 Bottom clause Construction Algorithm in YAVFORTE

Input: The current theory H' and the FDT, a clause C, Ex, an instance
Output: The Bottom Clause BC

1: **if** the predicate in head of C is different from the predicate in Ex **then**
2: $Ex \leftarrow$ fabricated instance (Muggleton et al., 2010a)
3: **if** C has an empty body **then**
4: $\perp \leftarrow$ Usual Bottom Clause Construction (Muggleton, 1995), with input $H' \cup FDT$ and Ex
5: **else**
6: $InTerms \leftarrow \emptyset$, $\perp \leftarrow \emptyset$
7: $C_{ground} \leftarrow$ instantiation of the clause C using H', FDT, and Ex, with substitution θ maintaining the variables of C
8: **for** each v/t in θ **do**
9: $InTerms \leftarrow InTerms \cup t$
10: $\perp \leftarrow \perp \cup C$
11: $i \leftarrow 0$, corresponding to the variables depth
12: $BK \leftarrow FDT \cup Ex$
13: **for each** modeb declaration b **do**
14: **for** all possible substitution θ of arguments corresponding to $+$ type by terms in the set $InTerms$ **do**
15: **repeat**
16: **if** b succeeds with substitution θ' **then**
17: **for each** v/t in θ and θ' **do**
18: **if** v corresponds to $\sharp$ type **then**
19: replace v in b by t
20: **else**
21: replace v in b by v_k, where $k = hash(t)$
22: **if** v corresponds to $-$ type **then**
23: $InTerms \leftarrow InTerms \cup t$
24: **if** $b \notin C$ **then**
25: $\perp \leftarrow \perp \cup b$
26: **until** reaches recall times
27: $i \leftarrow i + 1$
28: Go to line 14 if the maximum depth of variables is not reached
29: **return** $\perp$.

in the original FORTE system, in the present paper they are built upon YAVFORTE, including the bottom clause and the mode declarations to bound the search space of possible clauses.

Next, the strategies for revising theories through SLS are devised. Firstly, Sect. 4.1 brings the stochastic algorithm developed to search for the revision points. Next, a number of SLS algorithms are presented for deciding which revision operator will be responsible for modifying the theory in Sect. 4.2. Finally, stochastic algorithms applied within the revision operators, for choosing literals to be added to or removed from a clause, are devised in Sect. 4.3.

4.1 Stochastic local search for revision points

Algorithm 5 is an abstraction of Algorithms 3 and 2, presenting the key steps that YAVFORTE follows when generating revision points (in line 6 of Algorithm 1).

The relevant examples are those ones whose provability can be affected after proposing some revision in that point.

It can be seen from Algorithm 5 that there are two major factors increasing the cost of searching for revision points: the set of examples and the size of the theory. This happens because *each* example must be tested on the theory, either to identify faulty clauses and/or literals or to check whether the example is relevant to the revision point. Moreover, each clause

Algorithm 5 Generating Revision Points

1: Identify the misclassified instances
2: $RPs \leftarrow$ clauses and/or antecedents responsible for misclassified instances
3: compute potential for each $rp \in RPs$
4: Identify relevant examples for each revision point
5: sort RPs by potential;

in the theory may be tested as a potential revision point. In this work, the cost burden by those tests is reduced by introducing a stochastic component within the search for revision points. Rather than always looking for all revision points in the theory, the stochastic component allows only a subset of that group to be sought.

The strategy developed does not only avoid searching in the whole theory for revision points but also avoids considering all misclassified examples. It works by alternating between stochastic and complete moves according to a certain probability, a strategy that can be seen as an instance of the Randomizing Iterative Improvement method. Thus, with a probability p_{rp} the stochastic move is taken and the procedure will look only for a subset of all the faulty points in the theory. The size of this subset is pre-defined by the user, and it indicates how many specialization and generalization revision points will be returned by the procedure. To guarantee that both types of revision points will actually be returned, the algorithm first searches for the specified number of specialization revision points, followed by the search for generalization revision points. With a probability of $1 - p_{rp}$, a complete move is taken, just as in the original algorithm.

The stochastic move works as follows to gather the subset of revision points. First, it selects a single misclassified example e^- at random, from the set of misclassified negative instances. Then, revision points are collected from e^-. In case e already produces the required number of revision points k, the search for specialization revision points stops. If it produces more than k, k points are chosen at random. If the instance does not have enough revision points, the procedure proceeds to collect more revision points by choosing another misclassified example e'^- at random. The same is done for generating k generalization revision points.

After collecting the subset of random revision points, it is time to find out the relevant examples. This is necessary to compute the potential and also to consider only those examples to be proved again when evaluating a modification on the revision point. Note that if the probability p_{rp} is 100 % and $k = 1$, the approach employed here reduces to a relational version of Rückert and Kramer (2003). Algorithm 6 exhibits the procedure for collecting revision points using a stochastic component, modified from Algorithm 5.

Time complexity of the original procedure is bounded by the number of training examples times the size of the theory. Stochastic moves have time complexity limited by k. Note that in the complete case, the theory is traversed even for correctly classified examples, because FORTE needs to collect the relevant examples. Although the stochastic search still has to find the relevant examples for the revision points in the set of all training examples, this is also a reduced space, since it is restricted to the set of revision points instead of traversing the whole theory. Additionally, in case the search returns only one kind of revision points, then either (1) the revision points are generalization ones and only failing examples must be considered (true and false negative) or (2) the revision points are specialization ones and only provable examples (true and false positive) need to be considered.

Algorithm 6 SLS Algorithm for generating revision points

Input: A set of positive and negative examples E, divided into the set of correctly classified examples, ECC and the set of incorrectly classified examples, EIC, Theory T, probability p_{rp}, an integer k
Output: A set of revision points RPs

$SpecRPs \leftarrow \emptyset$
$GenRPs \leftarrow \emptyset$
with probability p_{rp} **do**
  while $\#SpecRPs < k$ and there is at least one provable negative instance **do**
    $ex \leftarrow$ a misclassified negative instance chosen at random from EIC^-
    $num\_rp = k - \#SpecRPs$
    $RP_{ex} \leftarrow$ at most num_rp revision points generated from ex
    identify relevant examples for RP_{ex}
    $SpecRPs \leftarrow SpecRPs \cup RP_{ex}$
  while $\#GenRPs < k$ and there is at least one unprovable positive instance **do**
    $ex \leftarrow$ a misclassified positive instance chosen at random from EIC^+
    $num\_rp = k - \#GenRPs$
    $RP_{ex} \leftarrow$ at most num_rp revision points generated from ex
    identify relevant examples for RP_{ex}
    $GenRPs \leftarrow GenRPs \cup RP_{ex}$
  $RPs \leftarrow GenRPs \cup SpecRPs$;
otherwise
  execute Algorithm 5
return RPs

4.2 Stochastic local search for revisions

YAVFORTE encompasses two specialization and four generalization revision operators. Depending on the amount of revision points, there are going to be several possible revisions. Thus, the revision process can also benefit from stochastic search when applying revision operators. Additionally, the revision can also take advantage of stochastic local search techniques to escape from local maxima. When following a stochastic strategy to propose and implement revisions, there are different decisions to be made:

1. how the space of a candidate hypothesis is populated;
2. how the revision to be implemented is chosen; and
3. how to stop proposing revisions (stopping criteria).

To populate the space of candidate hypotheses (proposed revisions) and choosing the revision to be implemented, there are two possibilities:

1. **Greedy space** All appropriate revision operators are proposed at each revision point, exactly as the original Algorithm. All revisions composing the search space of candidate hypotheses are scored and the best revision is indeed implemented in the current theory.
2. **Randomized space** the space is composed of tuples containing the name of the revision operator and the revision point. We randomly select k such tuples. In this case, the revisions are not proposed beforehand, as only k need to have their score computed. In other words, It is sufficient to *enumerate* the possible revisions for all revision points. For instance, regarding a single specialization revision point SRP and assuming both revision operators are employed in the revision process, we will have two tuples in the list: one containing $<SRP$, add-antecedent$>$ and another containing $<SRP$, delete-rule$>$.

Assuming that we use $k = 1$, when using randomized space a single tuple will be chosen at random. As the revision has not been proposed yet, it is necessary to check if it is really

possible to modify the theory with that revision. In other words, it may be the case the chosen revision cannot produce any modification in the theory. This is the case, for example, when trying to delete a rule and it is the last one for a KB predicate, or still if is not possible to delete or add antecedents to the clause. Also, notice that we may want to test whether the actual revision is acceptable. If either these tests fail, another tuple is chosen, until there no more tuples are left.

In this work, we considered two possibilities to accept a chosen revision, taking into account that one would like to reach a theory better than the original one:

1. We require an improvement on the score.
2. We allow bad moves, but to accept them, a perturbation function is considered (Hoos and Stützle 2005). We use $score + 0.5 * (Potential + score)$, where $Potential$ is the number of examples that has indicated a necessity of revision at a given point.

Finally, the stopping criteria may be one of the following:

1. **Empty list** no more possible revisions can be implemented;
2. **Maximum score** the revision reaches a maximum score on the training set (for example, in case the evaluation function is accuracy, the maximum score could be 1.0);
3. **Number of steps** the iterative procedure already performed a maximum number of iterations;
4. **Worse score** The score of the candidate revision is worse than the current score.

We designed three stochastic versions of the top level algorithm of YAVFORTE. They implement different approaches in this design space:

1. Stochastic Hill-Climbing with random walk.
2. Stochastic Hill-Climbing with stochastic escape.
3. Stochastic greedy with random walk.

The first two strategies combine Stochastic Hill Climbing and Randomized Iterative Improvement techniques, by alternating between greedy and stochastic moves, based on a fixed probability, and requiring in some cases an improvement on score (as Hill-Climbing does). The difference is, the first algorithm always requires an improvement in the score, no matter whether the move is greedy or stochastic, while the second algorithm demands a better score only in greedy moves. When the move is stochastic, a bad move can be selected, but it is anyway constrained by the perturbation function. The third strategy also alternates between greedy and stochastic moves but it does not demand an improvement in the score in either case, since a revision only requires the application of the perturbation function .

To summarize, the decisions followed by each algorithm are described in Table 1.

Algorithm 7 comprises the devised strategies, as all of them perform random walks, alternating between greedy and stochastic moves.

4.3 Stochastic local search for literals

The Add-antecedents operator introduces goals in a clause in order to stop proving negative examples while covering as many of the originally proven positive examples as possible. FORTE either uses a Hill-Climbing procedure, where at each iteration the antecedent which improves the score the most is chosen to be added in the clause, or it uses the *relational pathfinding* algorithm, where more than one antecedent can be added to a clause at once. These two approaches can also be combined, with the relational pathfinding algorithm being executed and, next, antecedents being added to a clause through the Hill-Climbing algorithm.

Table 1 Decisions followed by stochastic algorithms used to search for revisions

Decision	Generating space of candidate hypotheses and choosing a revision	Accepting a revision to be implemented	Stopping criteria
Algorithm			
Stochastic Hill-Climbing with random walk (SHC)	With probability p, randomized, otherwise, greedy	Improvement is always required	Empty list, maximum score, worse score
Stochastic Hill-Climbing with stochastic escape (HC-escape)	With probability p, randomized, otherwise, is greedy	Requires improvement if the move is greedy; otherwise, follows perturbation function	Empty list, maximum score, number of steps
Stochastic greedy with random walk (SGreedy)	With probability p, randomized, otherwise, is greedy	Follows perturbation function	Empty list, maximum score, number of steps

Algorithm 7 Stochastic Revisions Algorithm Based on Random Walks

Input: An initial theory T, A Background Knowledge FDT, a set of examples E, integer $maxSteps$, real $maxScore$
Output: A revised theory T'

```
score ← compute score of T
repeat
  generate revision points
  with probability p_rev do
    possibleRevisions revisions enumerated from the revision points and respective revision operators
    repeat
      nextRevision ← a revision chosen at random from possibleRevisions
      T' ← T after implementing nextRevision
      if an improvement in score is required then
        scoreNextRevision ← score of T'
        if scoreNextRevision > score then
          T ← T'
          score ← scoreNextRevision
        else
          possibleRevisions ← possibleRevisions − nextRevision
      else
        if a perturbation function is used to accept revisions then
          scoreNextRevision ← score of T'
          if scoreNextRevision + 0.5 * (Potential + scoreNextRevision) > 0 then
            T ← T'
            score ← scoreNextRevision
        else
          T ← T'
    until T' = T or possibleRevisions = ∅
  otherwise
    generate all possible revisions from the revision points and respective revision operators
    compute score scoreNextRevision of each proposed revision
    nextRevision ← revision with the highest score
    if an improvement in score is required then
      if scoreNextRevision > score then
        T ← implements nextRevision on T
        score ← scoreNextRevision
    else
      T ← implements nextRevision on T
until stopping criteria is matched
```

YAVFORTE restricts antecedents to the ones in a bottom clause generated from a covered positive example.

Similarly, the delete antecedents operator aims at making the clause prove positive examples while still not proving as many of the negative examples as possible. To achieve its goal, this operator either removes one antecedent at a time from the clause, using a Hill-Climbing approach, or it can delete multiple antecedents at once to escape from local maxima. The later approach is only used when the former does not produce any results, since it is expensive to list and test the combination of all possible literals to be removed from the clause. Both approaches require the modes language to be obeyed after a removal.

Regarding addition of antecedents, three main factors impact the search space of the bottom clause: (1) the size of the intentional and extensional background knowledge, (2) the number of different *modeb* definitions to the predicates together with the recall of each one of them, and (3) the setting of the variable depth parameter. In fact, as it was shown in Muggleton (1995), the cardinality of a bottom clause is bounded by $r(|M|j + j-)^{ij^+}$,

where $|M|$ is the cardinality of the set of mode declarations, j^+ is the number of $+$ type occurrences in each $modeb$ in M plus the number of –type occurrences in each $modeh$, j^- is the number of –type occurrences in each $modeb$ in M plus the number of $+$ type occurrences in each $modeh$, r is the recall of each mode $m \in M$, and i is the maximum variable depth. Thus, the bottom clause generates a search space of exponential size w.r.t. the maximum variable depth. In case the recall r is defined as $*$, which is the most common case, all the possible instantiations of a literal are going to be collected in the BK. Because of that, the size of the BK also influences the cardinality of the bottom clause.

Each element of the bottom clause may be tested on each relevant example to the clause being specialized, excluding those which does not have variables compatible with the mode declarations of the clause. In the worst case, to specialize a single clause it is necessary to pick up literals from the bottom clause as many times as the maximum size set of the clause. Given all these factors, addition of antecedents is an expensive operation and still performed many times during the whole revision process. Therefore, we propose to make the add antecedents operator, and consequently the revision process, more efficient by introducing stochastic components on this. Once again, we sacrifice completeness to gain efficiency when proposing modifications on the theory by adding antecedents to clauses or creating new rules.

Deleting antecedents is a less expensive operation than adding antecedents, as the search space only consists of the clause literals. Note that sometimes the search space can be less than the size of the clause, because removing some literals can make the clause illegal on the mode language. However, they still increase the running time, although slightly, since it is necessary to check if they can or cannot be a candidate to be deleted. Additionally, in the worst case, the operation of deleting antecedents may be performed $|clause|$ times in case deletions always improve the score. Thus, although benefiting less than when adding antecedents, we can also improve running time of the proposals of modifications by making the delete antecedent operator more efficient. Aiming toward this goal, we also introduce stochastic search when deleting antecedents, either when proposing generalizations on a single clause or when generalizing the theory by creating a new rule from an existing one.

Stochastic versions of the delete antecedents and add antecedents algorithms were developed according to the following strategy. They may perform either a random or a greedy move, depending upon a fixed probability. While the greedy move is the same for both approaches, since, in this case, the original algorithm is maintained, the random move differs from each approach. Next we devise each approach separately.

4.3.1 Stochastic component when searching for literals

We follow a stochastic Hill-Climbing approach and adopt a conservative strategy even when performing a random move, by requiring improvement of the evaluation function. The stochastic component is employed for choosing the next candidate clause. Basically, a random walk is carried out by taking a random step for choosing the next candidate clause with a fixed probability p_l, In case the probability p_l is not achieved, the original greedy hill climbing algorithm is performed. This algorithm is built upon the following decisions.

– **Defining the search space** In case the move is random and the goal is to add single antecedents, a literal picked at random from the bottom clause is added to the current clause to form a candidate clause. Similarly, a path created from the literals of the bottom clause in relational pathfinding algorithm is chosen at random. To delete antecedents from a clause in a stochastic move, the body of the clause is randomized and then a literal is chosen. In a similar way, combinations of literals are randomized and one of them is

taken at random. In all these cases, the candidate clause must be valid according to the mode declarations.
- **Choosing the next clause** we conservatively require that the next clause improves the current score, and choose the first candidate clause that is able to do that. Thus, if an evaluated clause does not improve the score, this procedure repeats until finding a candidate clause improving the score or exhausting the search space. The exception is the relational pathfinding algorithm, since the procedure allows a path to be chosen if the score is unchanged, as hill climbing can be employed to further specialize the resulting clause.
- **Stopping criteria** As usual in Hill-Climbing approaches, the algorithm stops when there are no more candidate clauses improving the score, either because the set of generated candidate clauses are not able to do that, or because there are no further valid clause to be evaluated.

Algorithm 8 replaces Hill-Climbing addition of antecedents in both add-antecedent specialization operator and in the second phase of the add-rule generalization operator. The algorithm starts by generating the bottom clause from a covered positive example, as it is done in the original Algorithm. Next, it performs a random walk, following the approach of algorithms such as WalkSAT, and decides the type of the move, based on a fixed probability p_{ls}. In case p_{ls} is not reached, the algorithm performs a greedy Hill-Climbing step, exactly as it done in the original algorithm: all valid (according to modes) candidate clauses formed by adding the literals from the bottom clause to the current clause are evaluated on the examples. Then, the candidate clause improving the score at most is selected. If there is no such improving clause, the procedure returns nothing. If the probability p_{ls} is reached, a random step is taken: a literal is selected at random from the bottom clause and added to the current clause. After the candidate clause is validated relative to the modes, it is evaluated using the examples. In case such a clause improves the current score, it is chosen to replace the current clause. Otherwise, it is discarded and another candidate clause is selected. This procedure continues until it finds a clause that improves the score or until it exhausts all the possibilities. Finally, the candidate clause replaces the current clause (if there is one) and the algorithm proceeds to the next iteration. This procedure is performed until there is no further clause improving the score or if it reaches the maximum size defined to clauses.

Relational pathfinding algorithm provides a sequence of antecedents to be introduced in a clause. The algorithm searches for all possible sequences and chooses the one with the highest score. In case of a tie, the smallest sequence is chosen. A stochastic version of this algorithm selects the sequence to be added to the current clause according to a stochastic decision: with a probability p_{ls} it chooses a sequence at random from all the possible generated paths; otherwise it proceeds as in the original algorithm. We do not generate paths at random, since this algorithm tries to find a sequence of literals that connect the variables in the head of the clause. Introducing a randomness component into this process could either disregard a possible valid sequence of literals or force the procedure to backtrack to several previous points. As it was said before, this algorithm is quite expensive by itself and, therefore, introducing more backtracks goes contrary to our primary objective of reducing the running time. Therefore, the benefit that the stochastic algorithm brings is to avoid the heavy computation of scores that considers the set of examples for each possible sequence. Algorithm 9 details the complete procedure.

The delete-antecedent operator benefits less from stochastic local search than add-antecedent, since the search space is restricted to goals in the clause and is, therefore, much smaller. The Hill-Climbing stochastic algorithm for antecedent deletion is shown in Algo-

Algorithm 8 Algorithm for adding antecedents using Hill-Climbing SLS

Input: A clause C, the maximum size of a clause, CL, the probability of deciding which move is going to be taken p_{ls}
Output: A (specialized) clause C'

repeat
  $currentScore \leftarrow$ compute score of C;
  $BC \leftarrow$ createBottomClause(...);
  with probability p_{ls} **do**
    repeat
      $ante \leftarrow$ an antecedent chosen at random from BC, whose input variables are already in C (therefore it obeys modes);
      $C' \leftarrow C$ with $ante$ added to it;
      $FPC \leftarrow$ negative examples whose proof considers the clause C;
      $candidateScore \leftarrow$ score of C';
      if $candidateScore > currentScore$ **then**
        $C \leftarrow C'$
        $currentScore \leftarrow candidateScore$
      else
        $BC \leftarrow BC - ante$
    until $C = C'$ or $BC \neq \emptyset$
  otherwise
    for each antecedent $ante \in BC$ **do**
      $C' \leftarrow C$ with $ante$ added to, in case $C + ante$ obeys the mode declarations;
      $candidateScore$ score of C';
    $bestClause \leftarrow$ candidate clause with the highest $candidateScore$
    if $candidateScore > currentScore$ **then**
      $C \leftarrow bestClause$
      $currentScore \leftarrow candidateScore$
  remove $ante$ from BC
  $FPC \leftarrow FPC-$instances in FPC not proved by C;
until $FPC = \emptyset$ or there are no more antecedents in BC or it is not possible to improve the score of the current clause or $|C| = CL$
return C

Algorithm 9 Stochastic Relational-pathfinding

generate all possible sequence of antecedents through relational_pathfinding algorithm and the Bottom clause;
with probability p_{l2} **do**
  choose a sequence at random;
otherwise
  choose a sequence with the highest score or the one with fewer antecedents in case of a tie;

rithm 10. Notice that delete-antecedent is also part of add-rule, with the latter using it in its first phase. The algorithm follows exactly the same random walk approach as previous algorithms seen in this section. First, it decides which type of move it is going to take, namely, a greedy move or a random move, based on a fixed probability p_{lg}. In case the move is greedy, it uses the original algorithm to propose deletions of antecedents. Otherwise, it selects at random a literal to be removed from the clause. Both cases require an improvement in the score to indeed remove a literal from the clause.

Algorithm 10 Algorithm for deleting antecedents using Hill-Climbing SLS

Input: A clause C, the probability of deciding which move is going to be taken p_{lg}
Output: A (generalized) clause C

```
1: repeat
2:    currentScore ← compute score of C;
3:    antes ← antecedents from the body of C;
4:    with probability p_lg do
5:       repeat
6:          ante ← an antecedent chosen at random from antes, whose removal from C still makes it valid
            relative to modes;
7:          C' ← C with ante deleted from it;
8:          candidateScore ← compute score of C';
9:          if candidateScore > currentScore then
10:            C ← C'
11:            currentScore ← candidateScore
12:         else
13:            antes ← antes − ante
14:      until C = C' or antes = ∅
15:   otherwise
16:      for each antecedent ante ∈ antes do
17:         C' ← C with ante deleted from;
18:         candidateScore ← compute score of C';
19:      bestClause ← candidate clause with the highest candidateScore
20:      if candidateScore > currentScore then
21:         C ← C'
22:         currentScore ← candidateScore
23: until no antecedent can improve the score;
24: return C
```

5 Experimental results

Stochastic algorithms usually provide two benefits: (1) reduction of the runtime, as completeness is abdicated in favor of randomness, and (2) the possibility of escaping from local maxima, since sometimes a move taken by an SLS algorithm is not greedily chosen. Thus, we would like to mainly investigate in this paper if it is possible to achieve such benefits when applying SLS algorithms in the theory revision process. We compare the deterministic approach of the revision system and a standard ILP system to each stochastic algorithm devised in this work, regarding the accuracy and running time. In addition, we also would like to verify whether theory revision is actually capable of providing better accuracies compared to learning from scratch, in a feasible time.

To investigate these questions, we designed two separated sets of experiments. The first one relies on the revision of a human crafted theory. The second one tries to revise theories generated from an ILP system. In this last case, we experiment in two ways: (1) the learning system gets a reduced amount of the training set while the revision system gets all of them, and (2) both systems get the same training set. Next sections we present the details and results of each set of experiments.

5.1 Human-engineered theory revision

We considered the UW-CSE domain to revise a human-crafted theory, since a number of clauses written by people at the University of Washington is provided, together with facts

and positive examples (Richardson and Domingos 2006). This domain contains facts related to the Department of Computer Science and Engineering of the University of Washington. The examples represent the relationship $advisedby$ between a professor and a student. The dataset used in this paper is composed of 113 positive instances, 2711 negative instances, 5082 ground atoms and 43 definite clauses, from which 8 of them have the target predicate $advisedby$ in their head, and the rest of them are intermediate clauses. As intermediate clauses can also be selected as revision points, it is essential to have the procedure that fabricates instances described in Sect. 3.3.1.

We performed three main changes in the set of provided clauses to make it possible to apply the revision system on them:

1. A clause with more than one positive literal was split into two clauses, as YAVFORTE is only able to deal with definite clauses.
2. In case the target predicate $advisedby/2$ appeared as a negative literal in a clause, it would be removed from the body, to avoid recursion.
3. Clauses with mutual recursion were removed from the set of clauses, otherwise it could become very hard to prove some examples without causing an overflow in the stack of the inference engine.

Experimental Methodology This dataset is already provided with five disjoint folds, where each one represents a Computer Science area of the University of Washington. Besides the revision system, we run Aleph (Srinivasan 2001) as the baseline learning ILP system. The m-estimate evaluation function (Dzeroski and Bratko 1992) was used as the optimization function, both in YAVFORTE and Aleph. In order to compare the performance obtained by the algorithms, we use F-measure, since this is a highly skewed dataset. All the other parameters were left at their default values, except for $minpos$ and $noise$, which were set to 2 and 100, respectively. Without the first one, Aleph would generate only a single literal clause for each positive example. As the target predicate is binary, relational pathfinding algorithm is the default choice to specialize clauses in YAVFORTE.

Concerning the stochastic algorithms, we considered two possible values of probabilities for pursuing a random move, which are 50 and 100 %. In the last case, a random move is always performed. Each stochastic algorithm was executed 5 times, to diminish the possibility of getting a result simply because of luck (or out of it). Next, we present the results for the deterministic and stochastic algorithms. The plots show the average values obtained from the runs and also an error bar representing the standard deviation, computed over each run.

All the experiments were run on YAP Prolog (Santos Costa et al. 2012) on Power Edge R420 Intel Xeon-12 cores RAM 16 GB HD 500GB Net 1 GB/s machines. In order to verify statistical significance, we used the two-tailed paired t test with $p < 0.05$.

5.1.1 Results

Deterministic Approaches Figure 1 exhibits the results of predictive precision and recall for each fold in the UW-CSE dataset, concerning the deterministic version of YAVFORTE, Aleph system and the values collected from the human-crafted theory. Note that the recall is 1.0 in the initial theory, since all positive examples are correctly classified, while the precision is very low, as all of the negative examples are also proved. This indicates that the theory needs to be primarily specialized. The other implication of this fact is that when we include the initial theory in the Background Knowledge provided to Aleph, it has nothing to do, as its default algorithm construct clauses starting from unprovable positive examples, and

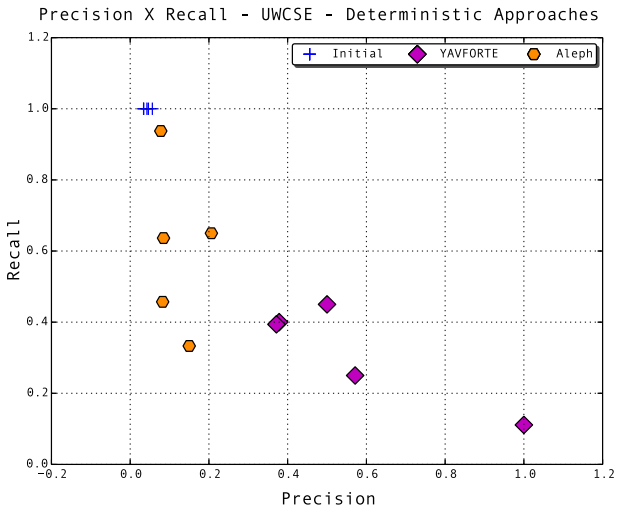

Fig. 1 Scatter plot with precision versus recall, where each point represents the values obtained from each one of the fivefolds of the UWCSE dataset

in this case there is none. Because of that, Aleph only properly worked when the BK was composed with the 5082 facts and not with the human-crafted clauses. The same set positive and negative examples is used for all the approaches.

From the figure, we can observe that although Aleph reaches better values of recall, the precision is rather low, as a large number of negative examples remain covered. YAVFORTE exhibits almost an opposite behavior, as it reaches better precision than recall. As YAVFORTE is presented with a large number of negative examples, the revision process tends to specialize the initial theory more than generalize it. Because of that, it can yield a theory with less negative examples covered, compared to Aleph and the initial theory, in detriment of making some positive examples unprovable. However, while YAVFORTE is significantly better than Aleph concerning the precision values, there is no statistical difference between the recall values reached by both systems. This indicates that YAVFORTE is capable of providing a better trade-off between precision and recall.

This better trade-off between both measures can be better visualized in Fig. 2, that presents the F-measure and runtime results of each fold in the UW-CSE dataset. Note that we set the running time of the initial theory as 0, as there is no learning associated with it. YAVFORTE presents statistically better results of F-Measure than Aleph in this domain, and also executes in less time, which are consequences of specializing an initial theory, followed by alternating between specialization and generalization of faulty points, rather than learning from scratch.

Stochastic Approaches In order to verify the behavior of the stochastic algorithms, we execute each one of them varying its essential parameters. We compare their performance to the deterministic revision system and Aleph system. We start with the algorithm that randomizes revision points, which, instead of selecting all the revision points, with a certain probability chooses at random only a previously specified number of revision points. We set the probability to 50 and 100 % and the number of revision points to 1, 2, 4, 6, 8, 10. The real number of revision points is at most twice each one of these values, since we get the same amount of generalization and specialization revision points.

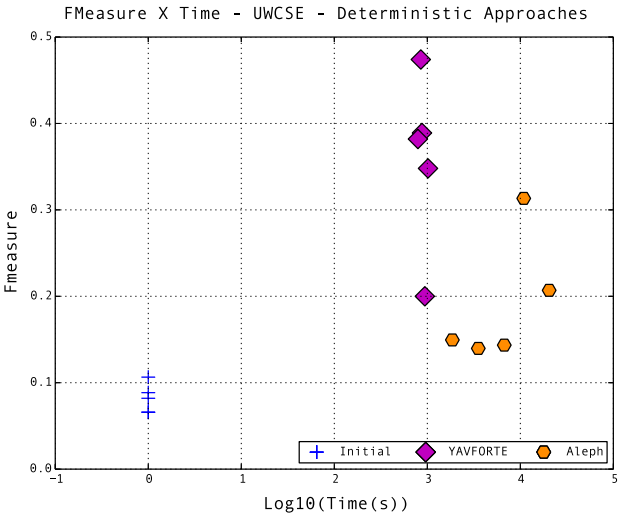

Fig. 2 Scatter plot with F-measure and runtime, where each point represents the values obtained from each one of the fivefolds in the UWCSE dataset. Time is represented in a logarithm-10 scale

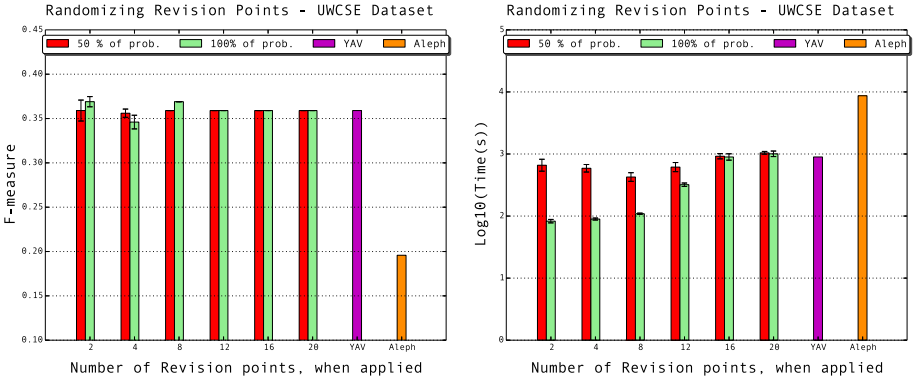

Fig. 3 Bar plot with the F-measure (*left*) and running time in logarithmic scale (*right*), obtained from the UW-CSE dataset, comparing different choices of number of revision points chosen at random and deterministic revision and learning

Figure 3 presents the F-measure and running time results computed for each version of the number of revision points parameter in the stochastic revision points algorithm, compared to deterministic YAVFORTE and Aleph. The bars of stochastic algorithms are annotated with the standard deviation collected from the 5 executions.

The most interesting result was achieved when the algorithm was set to always performing stochastic moves (probability of 100 % and to return at most 2 revision points at each iteration.[1] Note that, in this case, the stochastic version runs in more than one order of magnitude faster than the deterministic version, while even reaching better F-measure values, although not significantly better.

[1] We say it returns at most X revision points because it may be the case that there is no revision type of a type. For example, in the first iteration of the algorithm, there is no generalization revision point, since initially all positive examples are covered.

This domain has a relatively large number of clauses, where a good part of them are intermediate clauses, creating a highly nested theory. Finding out revision points in this kind of theory is an expensive task, as this process is based on generating all refutation (failure) paths in case of misclassified negative (positive) examples and annotate them. The whole process is usually more expensive than only trying to prove examples. Because of that, we can save a great amount of time when avoiding searching for all possible revision points. On the other hand, the initial theory has a limited base clauses that can, in fact, improve its score. The majority of clauses in this set are responsible for misclassifying a large number of examples. Because of that, when we choose an example at random, is highly probable that it will point out exactly the revision points that can improve the theory.

Although we do not exactly follow an incremental revision process here, as we do not keep changing the theory along the arrival of new examples, this result suggests that it is possible to improve theories even when considering only a single example to indicate the revision points. Albeit we bound the number of generated revision points, when this is only one, it is certain that we got it from a single misclassified example. This suggests that this stochastic method can be further investigated to act in domains where the examples arrive in streams (Gama 2010).

Considering the further results, we can see that by setting the random move probability as 50 % the F-measure is better or the same as the one obtained in the deterministic version. However, the speedup in running time is only obtained when we collect up to 6 revision points of each type. In the best of these specific cases, we can reduce the running time to half of the deterministic version. When the probability is defined as 100 %, we obtained a small decrease in the performance with 2 revision points of each type. However, with 4 revision points the stochastic version is slightly better than the deterministic approach. This variance is too small and it is probably due to the random moves in a space larger than the one of only 1 revision point of each type.

When collecting 8 or 10 revision points of each type, the stochastic algorithm practically gets the same results as the deterministic version. This happens because, on average, the deterministic version also finds about this same amount of revision points. Note, however, that even when defining more revision points than it needed, the performance of the stochastic revision is not worse than the one of the deterministic version.

Next, we present the results of the algorithms that randomize revisions. There are three of them in this work, where the number of iterations is the stopping criteria for two of them (SHC, the Stochastic Hill Climbing approach, stops only when it is not possible to improve the score anymore). This parameter also fixes the maximum number of revisions that are going to be implemented, as at most one revision is implemented at each iteration. Note that it is possible to pass through an iteration without implementing any revision, which happens in two situations: (1) the move is greedy and the best revision cannot improve the score, or (2) the algorithm is Stochastic Hill-Climbing Escape, the move is random but the selected revision does not fulfill the scoring criteria. We set the maximum number of iteration parameter as 1, 2, 4, 8, 12, 16 and 20, when applied.[2]

Figure 4 presents the F-measure and running time results regarding the Stochastic Greedy (*SGreedy*) and Stochastic Hill-Climbing Escape (*SHC-Escape*) with the different numbers of maximum implemented revisions, the Stochastic Hill-Climbing (SHC) and the results of the deterministic approaches, for comparison.

[2] Clearly, the algorithms can also stop when there are no more revision points available, but this is rarely the case. Specifically in this domain, the final theory never correctly classifies all the training examples.

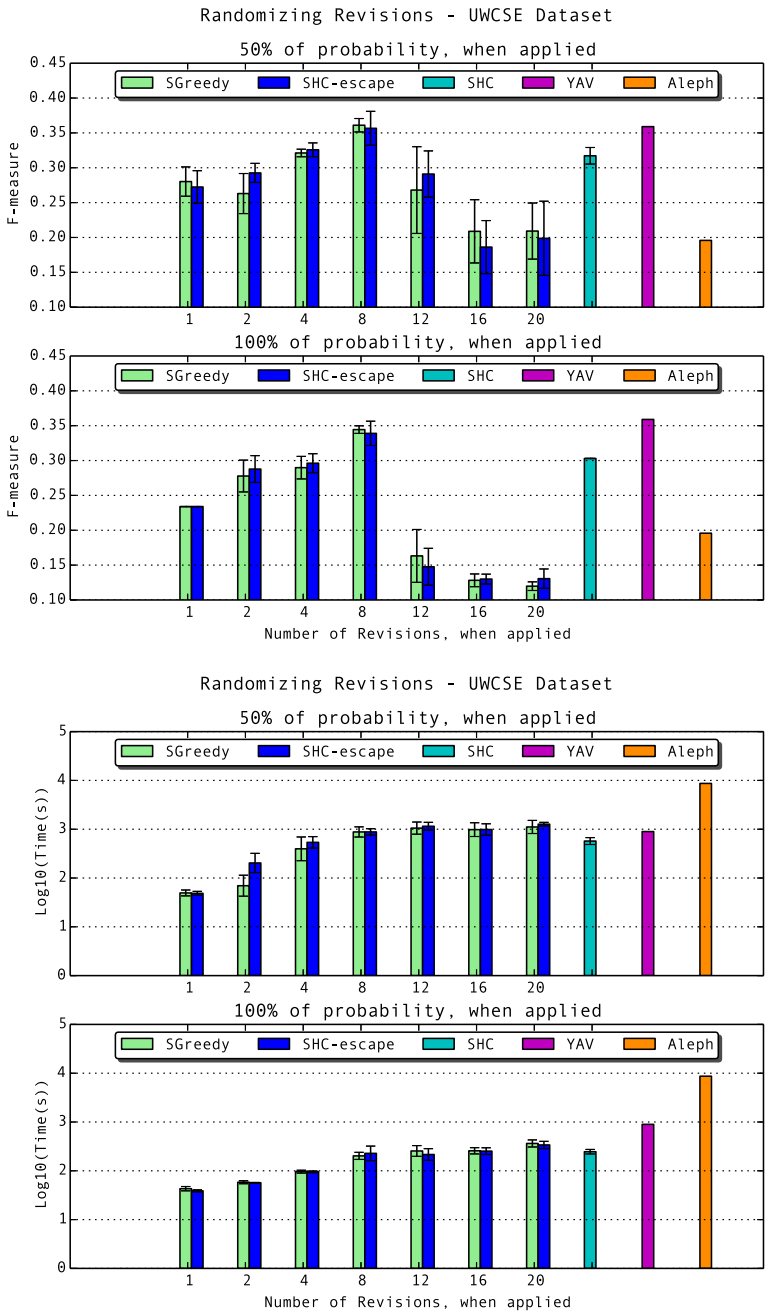

Fig. 4 Bar plot with the F-measure (*top*) and running time in logarithmic scale (*down*), obtained from the UW-CSE dataset. Here we visualize the results of the algorithms that randomize revisions, considering the parameters of the maximum number of iterations when this is the stopping criteria, and the deterministic revision and learning

On average, the deterministic version of the revision takes 6 iterations to finish the process, when it does not find any profitable revisions to be implemented. This indicates a possible number of ideally implemented revisions on this dataset, as the F-measure values obtained by SGreedy and SHC-escape start to rapidly decrease when the maximum iterations parameters are greater or equal to 12. Although this is the double of revisions implemented by the deterministic version, it is important to notice that the stochastic versions may improve the scoring function slower than the deterministic process, since they do not always select the best possible revision, due to the random nature of the algorithms. Because of that, observe that the Stochastic Hill Climbing approach, for example, cannot achieve the same performance as the deterministic Hill Climbing, while still does not reduce much the running time. In this specific case, setting the probability to 50 % makes the algorithm behave quite similar to the deterministic version. On the other hand, when the probability of taking a move at random is 100 %, the running time of the Stochastic Hill Climbing is 40 % the time of the deterministic version, while the F-measure is only a little affected by always choosing a revision at random.

The behavior of the SGreedy and SHC-Escape algorithms are quite similar, which is actually expected as they are both similar approaches, changing only when a greedy move is followed. However, in general, the SGreedy algorithm presents an unstable behavior, as its final performance varies both within the 5 runs and with the different numbers of maximum iterations.

We emphasize once again that the initial theory written for this domain is large, highly nested and contains only a small number of points that once revised makes a number of examples to become correctly classified. Thus, the greatest cost of the revision process is actually to find such revision points.

Finally, Fig. 5 presents the results obtained when randomizing literals to be added to or deleted from a clause. Surprisingly, this randomization was not able to reduce the running time, but actually it increased the total running time. This was caused by two reasons: (1) as any literal capable of improving the score can be added to the clause in a random move, in a number of attempts of specializations the algorithm took more time to converge, adding more literals than it was necessary up to reaching the maximum number of literals in a clause, and (2) relational pathfinding does not obey this limit of literals, as this could make the algorithm not finding a correct path. In the deterministic version, the best path is the one that covers more positive examples and, in case of a tie, the smaller one. The stochastic version just chooses the first path that is able to cover more than one positive example. Because of that, in several cases, the path chosen at random had more than five literals, which is the limit of literals in a clause when adding one literal at a time. This fact does not affect directly the searching for literals, but instead affect the cost of proving examples in the next steps of the revision process.

To sum up, Fig. 6 presents the average F-measure and running time results of the deterministic algorithms and of all the stochastic approaches, with different parameters. Furthermore, we include there the results of combining the stochastic approaches at each key step of the revision process. Thus, we execute together the randomization of the revision points (choosing at most five of them), the randomization of the antecedents and the randomization of the revisions with the stochastic Hill-Climbing approach. In order to also observe the impact of the probabilities when combining the stochastic approaches, we consider 50 and 100 % of probability for each strategy, yielding eight different values.

First, notice from the figure, that the best trade-off between running time and F-measure is achieved by the execution that only randomizes revision points and by the execution that combines the randomization at each key step of the revision process. In this last case, the

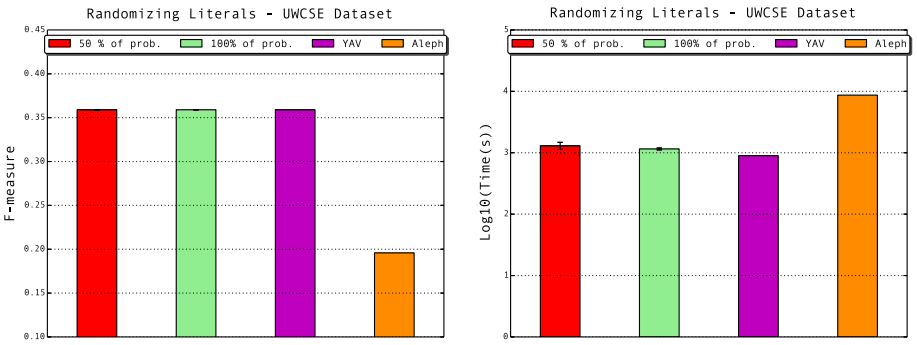

Fig. 5 Bar plot with the F-measure (*left*) and running time in logarithmic scale (*right*), obtained from the UW-CSE dataset. Here we visualize the results of the algorithms that randomize literals (or a path of them, in the case of relational pathfinding algorithm), and the deterministic revision and learning

running time is better than the former, and, although the F-measure values are slightly worse, there is no statistical difference among them.

Second, we observed that, in the vast majority of the cases, the F-measure results obtained from the stochastic approaches are not statistically different from the results of the deterministic version. On the other hand, even in the cases where the F-measure values are slightly worse than the deterministic revision, they are still usually better than the results obtained from Aleph. Furthermore, the running time of the stochastic approaches is often lower than both Aleph and deterministic YAVFORTE.

Third, on the negative side, observe that there is a number of cases with low running time but achieving F-measure results that are significantly worse than the deterministic revision, and equal or worse than the results obtained with Aleph (the ones below the 0.25 F-measure mark). Those are the runs where the choice of the parameters demanded more revisions than necessary. This is the case for Stochastic Greedy and Stochastic Hill-Climbing Escape with 16 and 20 as the maximum number of iterations. This is mostly due to the fact that the algorithms allowed bad moves and could not recover from them thereafter.

This is not the case for the randomization of revision points: even when we let the algorithm to select more revision points than the deterministic version could find, the performance of the algorithm is the same, as the real number of revision points is an upper bound, since it is not possible to return more revision points than in fact exist. Thus, in domains with characteristics similar to UW-CSE, when randomizing the revisions and using the number of iterations as stopping criteria, it is better to let the maximum number of iterations low than high. One would still get an improvement in the initial theory in this case, without taking risks of gaining nothing after the entire revision process.

5.2 Revising theories generated by an ILP system

In this section we experimentally investigate the behavior of the revision systems by simulating two scenarios: (1) an initial theory has been induced by a learning system, considering a certain amount of examples, and new examples arrive, that may require such theory to be adjusted; and (2) a theory has been induced by a learning system but it might be still room for improving it. To verify the behavior of the deterministic and stochastic revision procedure in these two scenarios, we once again use the standard learning system Aleph as the baseline inducer of the initial theories. Next, we briefly present the benchmark datasets that we have

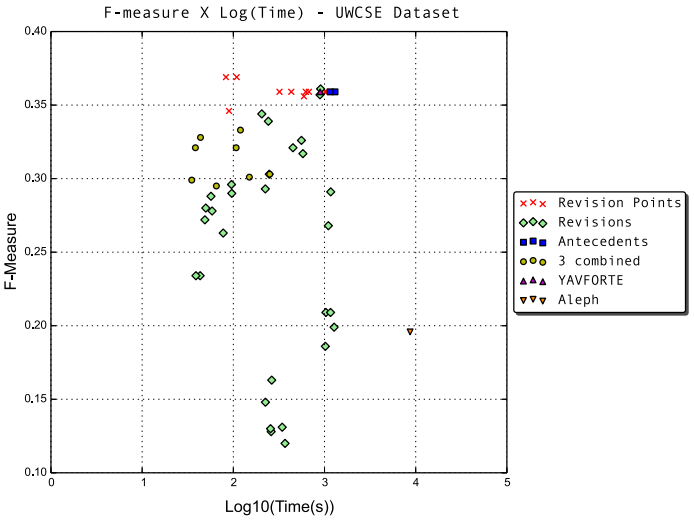

Fig. 6 Scatter plot comparing the running time and F-measure of the deterministic algorithms, the algorithms that randomize each search step individually, varying their parameters, and combinations of the algorithms that randomize each search step (3 combined, in the legend), varying their probabilities

used in the experiments, the methodology that was followed to simulate both scenarios and the results obtained from them.

Datasets We considered four domains in this work:

- Pyrimidines is a Quantitative Structure Activity Relationships (QSAR) problem, concerning the inhibition of E. Coli Dihydrofolate Reductase by *pyrimidines*, which are antibiotics acting by inhibiting Dihydrolate Reductase, an enzyme on the pathway to forming DNA (King et al. 1992; Hirst et al. 1994). The dataset we used in this work is composed of 2361 positive examples, 2361 negative examples, about 2200 facts in the Background knowledge and 28 templates of literals in the language bias. The target predicate is $great/2$.
- Yeast_sensitivity (Kadupitige et al. 2009) is a dataset concerning the problem of gene interaction of the *yeast Saccharomyces cerevisiae*. It is composed of 430 positive examples, 680 negative examples, approximately 170,000 facts and 18 templates of literals in the language bias, to yield literals to clauses. The target predicate is $general\_responder/1$.
- Proteins domain is a task of secondary structure protein prediction. The task is to learn rules to identify whether a position in a protein is in an alpha-helix (Muggleton et al. 1992), where the target predicate is . We considered a dataset with 1070 positives examples, 970 negative examples, about 5200 facts in the Background knowledge and 46 different template of literals in the language bias. The target predicate is $alpha/2$.
- Metabolism is based on the data provided by the 2001 KDD Cup (Cheng et al. 2002). The data consists of about 6900 ground facts about 115 positive instances and 115 negative instances. There are 10 possible templates of literals to be added to clauses. The target predicate is $metabolism/1$.

Experimental Methodology The datasets were split into 10 disjoint folds sets to use a K-fold stratified cross-validation approach. Each fold keeps the rate of original distribution

of positive and negative examples (Kohavi 1995). In order to simulate the arrival of new examples, we reduced the training set of the learning system to 40, 50, 60, 70 and 80 % of examples of the original training set. The revision system receives as input the theories generated from such sets and the whole set of training examples. In all these cases, the validation set was left as in the original folds, so that we would not mix up training and test sets in different executions. Also, a different theory was generated for each run of the cross-validation procedure. In addition, we would like to see if the revision system is able to improve the theory induced by the learning system when both have as input the same training set. Thus, we also generated a theory with Aleph with the whole set of training examples for each run of the cross-validation procedure and give such theories as input to the revision system, i.e., the same training set is used to learn and to revise the theory. Stochastic algorithms were run 5 times because of the random choices. The plots report the average values obtained from all folds and all runs and the standard deviation computed from the runs.

The initial theories were obtained from Aleph system using its default parameters, except for the five following: (1) $clauselength$, which is defined as 10, except for Yeast Sensitivity that took too long to run, and then we set such parameter to 5; (2) $noise$, defined as 50, (3) $nodes$ set to 10000, (4) $minpos$, set to 2 and rules were induced through (5) $induce\_cover$ command. The use of those parameters has been inspired on the work of Muggleton et al. (2010b). We used the default evaluation function of both systems and used the Accuracy to compare their performance. Concerning deterministic YAVFORTE parameters, besides the evaluation function, it has only the clause length parameter in common with the cited just before, which we let with the same value. Relational pathfinding was used in the datasets that have a binary target predicate. We do not argue that those are the best parameters neither for Aleph nor for the revision procedure. Tuning parameters of a system with several parameters as Aleph is not a trivial task and it is not essential to our experiments, as we would like to investigate the revision behavior with an ad-hoc received theory.

All the experiments were run on YAP Prolog (Santos Costa et al. 2012) on machines Power Edge R420 Intel Xeon—12 cores RAM 16 GB HD 500 GB Net 1 GB/s (first two datasets) and Dells Optiplex core i7—4 cores RAM 8GB HD 500GB Net 1GB/s (2GB per core) (last two datasets). In order to verify statistical significance, we used a two-tailed paired t test with $p < 0.05$.

5.2.1 Results

We first present the results of the revision process when revising theories generated with different training set sizes. Then, we compare the results of the stochastic approaches to the deterministic algorithms.

Deterministic Approaches In this case, we used all the settings of training sets to see how the revision system behaves when facing a theory that was not generated with all the examples that the revision itself has; and when facing a theory generated with exactly the same examples that it has.

Figures 7 and 8 exhibit the results of accuracy and running time for each setting of training set size. We present the accuracies obtained from the learning from scratch with a (possibly) reduced training set and the accuracies that the revision system achieves, always revising the initial theory with the whole training set. In addition, we present the results of Aleph when it gets the same input as the revision system, i.e., an initial theory obtained from the different

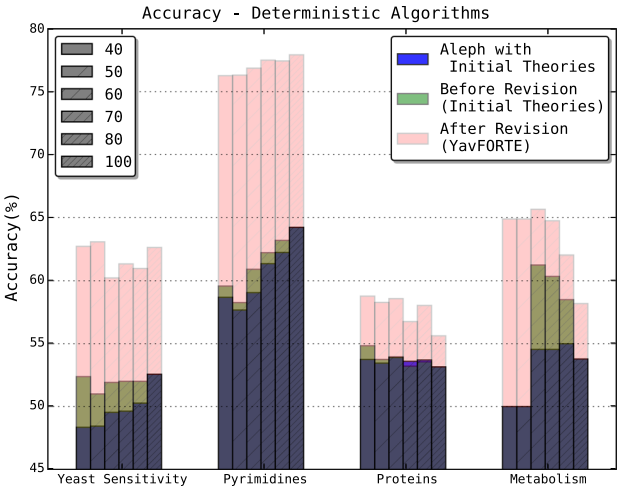

Fig. 7 Accuracy of deterministic revision and learning, for all datasets. A *whole bar* indicates the total accuracy obtained by a system. Different hatches frequencies represents the percentage of examples used to generate the initial theories

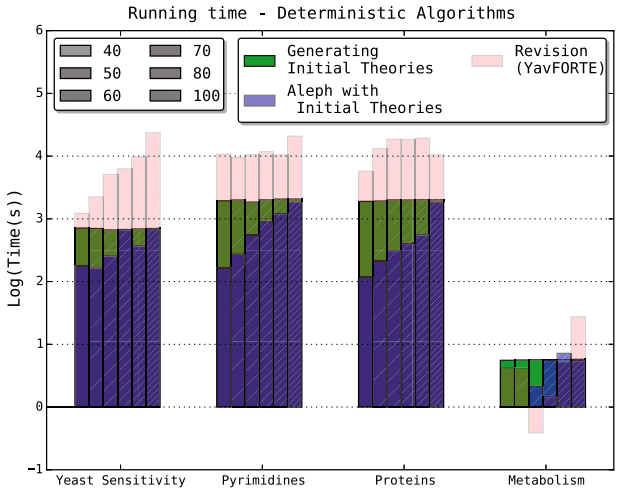

Fig. 8 Running time for generating initial theories (Aleph) and for revising them (YAVFORTE). A *whole bar* indicates the total time obtained by a system. Different hatches frequencies represent the percentage of examples used to generate the initial theories

settings of size for the training set and the whole set of training examples. The running time is computed for each process as well.

As expected, the revision system was always capable of improving the accuracies of the initial theories, even when the initial theory was built from the exactly same set of examples that the revision process received. This shows the efficacy of the revision process even as a sort of post-pruning procedure. Observe that, in general, the better is the initial theory, a higher accuracy the revision system reaches. In these results, the only extreme exception to that pattern happened in the Yeast_sensitivity dataset with 50 % of the examples generating

the initial theory, as the initial accuracy was the worst one and the final accuracy was the best one.

Note that in the most of the cases when providing Aleph with the whole set of examples and the initial theories obtained from a subset of them, the accuracies of the induced theories are either the same of worse than when considering only the initial theory. Aleph does not start from the initial theory in the background knowledge as the revision does, but only uses it in coverage tests. Because of that, such induced theories are quite similar to the one obtained by learning from scratch with the whole set of examples. In addition, notice that if the original clauses cover negative examples, they still will be provable after the learning process, what may diminish the final accuracies.

We do not guarantee that the initial theories get better following the number of examples. As can be seen, only Pyrimidines dataset had this behavior (the more examples the learning from scratch system has, the better is the initial theory). However, as the revision process got different results even that it considers the same set of examples to revise them, it is possible to conclude that they require different improvements. Also, notice that there are some cases where learning with the whole training set does not yield the best accuracy. This is the case of Proteins and Metabolism datasets. It could be the case that the learning system reached a local maximum that even the revision system could not escape from.

Furthermore, even when the learning from scratch system gets the same examples as the revision systems (the last column of each set of bars), the revision process still reaches better accuracies, no matter which set of examples generated the initial theory. In others words, if we compare the results that YAVFORTE achieves from theories originally learned from scratch with 40–80 % of the training set, while YAVFORTE itself sees the whole set of examples, to the results that Aleph achieves with the whole set of examples, the revision achieves better results. Such differences are almost always statistically significant, comparing both the improvement that the YAVFORTE system gets after revising the initial theories, and the accuracies that YAVFORTE and Aleph reach with the same set of examples. The exceptions are in the Metabolism dataset with 80 and 100 % of the training set. In these cases, neither the improvement is statistically significant nor the revision of the theory obtained from 80 % of the training is better than the theory induced by Aleph with the whole set of examples. Although, on average, the difference is quite high, this is due only to a few folds, which is not enough for the statistical test.

Finally, the learning process is always faster than the revision procedure, even when both systems receive the same set of examples. For the learning process, the more examples we provide, the higher is the running time. This is not the case of the revision process, as it always receive the same set of examples (the whole training set). Even though, we can still see that the running time of the revision smoothly increases, as the initial theories are larger according to the set of examples (eventually, more clauses are generated to cover more examples, and also the clauses got bigger to diminish the covering of negative examples). Note that Metabolism dataset did not follow this pattern. However, what happened here is that, with 40 and 50 % of the examples, the noise parameter was higher than the number of negative examples. Because of that, Aleph produced a trivial clause that covered all the examples. Also, with 60 and 70 % of the examples, the theories were too simple and the revision system did not have to explore a costly search space. Those are the only cases where the revision process was faster than the learning from scratch procedure.

We would like to emphasize that in this work we do not intend to simulate an incremental revision process (Esposito et al. 2000) that keeps revising a theory along with the arrival of new examples. Instead, we would like to observe the behavior of the revision process with ad-hoc initial theories. In an incremental revision, only the new examples would be

considered when revising the theory and an appropriate mechanism should be employed to do not make it stopping representing the previous examples. Such an incremental process could be very useful to learn streams of data (Gama 2010), especially in the presence of concept drift (Widmer and Kubat 1996; Gama et al. 2004).

Stochastic approaches We compare the stochastic algorithms set with a number of parameters with the deterministic approaches. Regarding the algorithm that randomizes revision points, we set the number of revision points as 1, 2, 4, 8, 16 of each type, thus limiting the number of returned revision points to be the double of these values. The maximum number of revisions implemented in a whole process of the algorithms that randomize revisions are established as 2, 4, 8, 16, 32.

Randomizing Revision Points Figures 9, 10, 11, and 12 shows the results of accuracies and running time when randomizing revision points. First of all, notice that the learning from scratch bounds below the majority of final accuracies, showing that even when the revision process chooses only one revision point of each type to propose modifications, it is still possible to improve the initial theory. This is particularly evident in Yeast Sensitivity dataset (Fig. 10).

On the other hand, the deterministic revision frequently bounds above the accuracies, although from 8 chosen revision points, the accuracies achieved by the stochastic approaches start to get quite close, and sometimes even better than, to the accuracies reached by the deterministic revision process. Note, however, that between 16 and 32 revision points the difference is quite small. In Metabolism dataset, which has the smallest initial theories, and consequently less revision points, choosing 32 revision points makes the stochastic algorithm to behave worse, as it loses its randomness component and achieves the same results as the deterministic revision process, regarding both probabilities. On the other hand, with only a total of 2 revision points, the stochastic algorithm already achieves better performance than the deterministic approach.

With regard to the probabilities, when it is 50 %, the results are slightly better, showing more difference when there are <8 revision points chosen at random. From this value, the differences in the accuracies obtained from both settings are often quite small. This suggests that when the number of pre-defined revision points is enough, there is no need of employing greedy moves and selecting all the possible revision points. Additionally, the number of

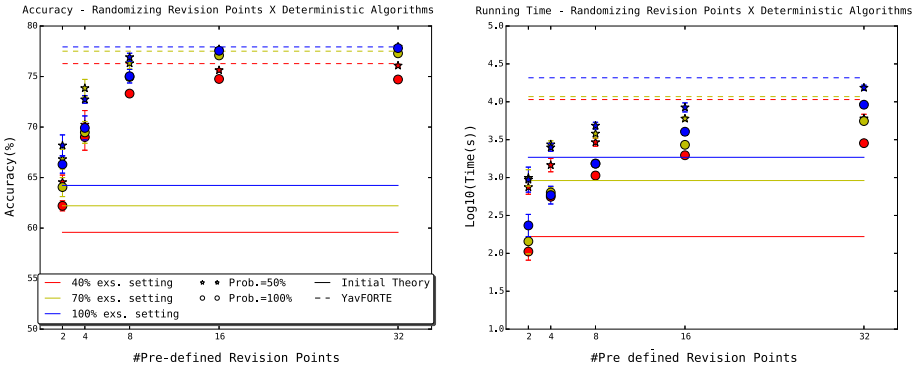

Fig. 9 Accuracy (*left*) and running time in logarithmic scale (*right*), obtained from the Pyrimidines dataset, when randomizing revision points. The legend is the same for both plots

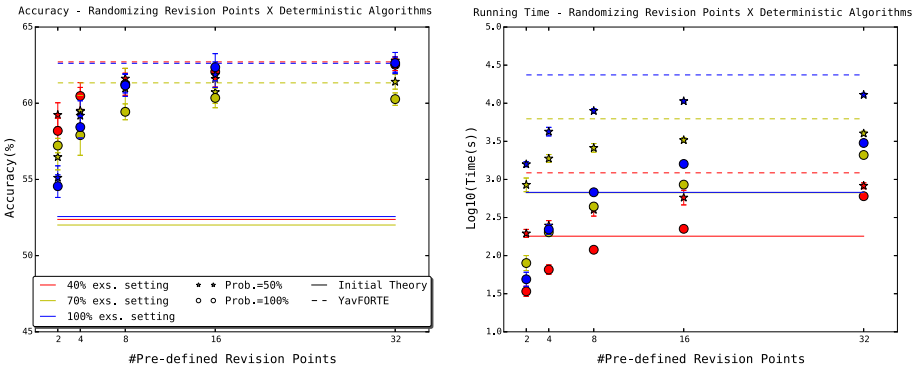

Fig. 10 Accuracy (*left*) and running time in logarithmic scale (*right*), obtained from the Yeast Sensitivity dataset, when randomizing revision points

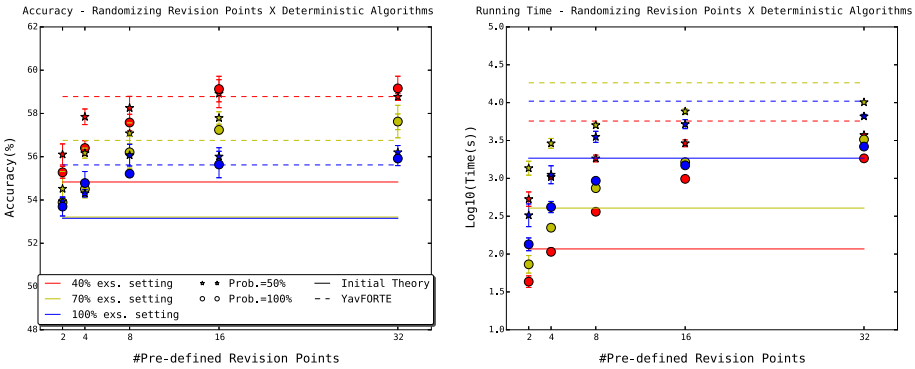

Fig. 11 Accuracy (*left*) and running time in logarithmic scale (*right*), obtained from the Proteins dataset, when randomizing revision points

pre-defined revision points follows the size of the initial theories, as in these experiments, the more clauses we have in the initial theory, the more revision points the revision system will probably find. Roughly speaking, the stochastic algorithm already achieves good results considering less than half the revision points that the deterministic revision finds.

Concerning the running time, as expected, the stochastic algorithm is faster when fewer revision points are considered. In some cases, it can be actually faster than the learning from scratch system. Notice that the learning from scratch system sees the same examples as the revision process only in the solid blue line and in a number of cases, this specific line is above the markers of the stochastic algorithm, even more when only random moves are considered (probability $= 100\,\%$). As the number of revision points grows, the running time of the stochastic algorithm approaches the ones of the deterministic revision.

Notice that there are cases where the stochastic algorithm can even reach better accuracies than the deterministic approach while running in a substantially reduced time (see, for example, the blue markers and dotted blue lines in Metabolism and Proteins datasets). In general, the accuracies of both approaches are competitive, usually when at least 8 revision points are considered, while the stochastic approach executes faster than the deterministic algorithm, and in some cases even faster than the learning from scratch system.

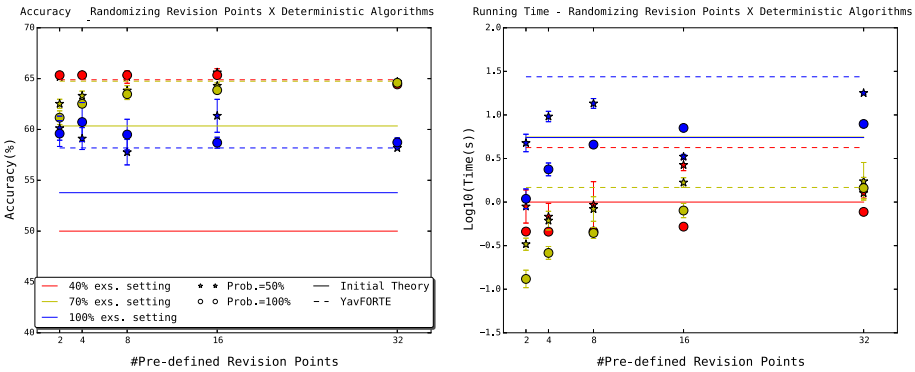

Fig. 12 Accuracy (*left*) and running time in logarithmic scale (*right*), obtained from the Metabolism dataset, when randomizing revision points

Randomizing Revisions Next, we present the results obtained from the algorithms that randomize the choice of the revision to be implemented in Figs. 13, 14, 15, and 16. Differently from the other stochastic algorithms, the Stochastic Hill Climbing (SHC) does not limit the number of implemented revisions (i.e. its stopping criteria is not the number of iterations of the overall revision process, but instead when it cannot find any revision that improves the current scoring). Because of that, it is represented in the figures similarly to the deterministic algorithms, with a dashed line. We limit the beginning and ending of this line with different symbols to differentiate the both probabilities used in the experiment.

Concerning the accuracies achieved by the Stochastic Hill Climbing approach first, we can see that the only dataset where the tree settings of initial theories and the both values of probabilities behave the same is the Pyrimidines dataset. In this specific case, the accuracies are always the same as the deterministic approach, while the running time of the former is lower than the later, reaching an order of magnitude for the 100 % of probability and examples generating the initial theory. This is the dataset that yields the larger theories. In the others datasets, the 100 and 70 % setting of examples usually present better cases for the stochastic approaches, while the 40 % setting is usually better for the deterministic approach. However, in all of these cases, the running time is greatly reduced, even being smaller than Aleph in the proteins dataset, 100 % setting and 100 % of probability. Notice that, as only 40 % of the examples are used to generate the initial theory, in this setting the theories are smaller than in the others cases. Also, observe that the smallest difference between the stochastic and deterministic algorithms appears in the Metabolism dataset, which is the one with the smallest initial theories. Finally, while the choice of probability makes a reasonable difference in the running time, it little influences the final accuracies. What happens here is that with the probability of 100 %, the algorithm takes more iterations to converge. However, as it always chooses the revisions at random, it still has smaller search spaces. Taking all those observations into account, these results suggest that the presence of large theories benefits this type of stochastic algorithm, as the search space of possible revisions may be also very large.

The size of initial theories also largely influences the results of both Stochastic Greedy and Stochastic Hill Climbing with Random Escape algorithms, but in this case it is more likely to be due to the stopping criteria, which is the number of implemented revisions. First of all, notice that both of them have similar behavior, and therefore it seems to make no difference to require an improvement in greedy moves, as it is the case of the second one.

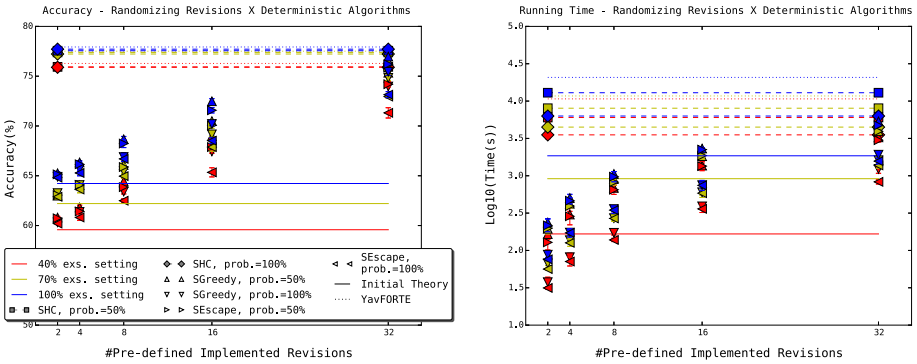

Fig. 13 Markers with the accuracy (*left*) and running time in logarithmic scale (*right*), obtained from the Pyrimidines dataset

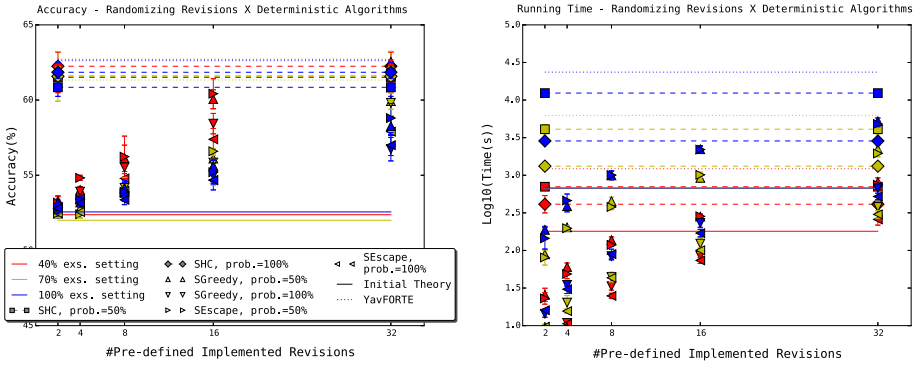

Fig. 14 Markers with the accuracy (*left*) and running time in logarithmic scale (*right*), obtained from the Yeast Sensitivity dataset

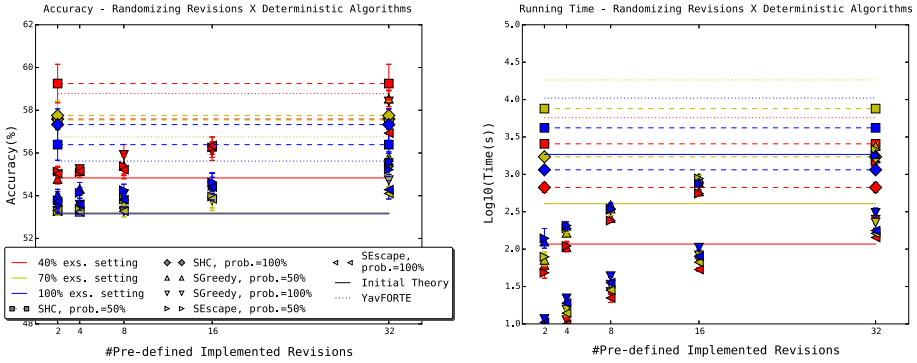

Fig. 15 Markers with the accuracy (*left*) and running time in logarithmic scale (*right*), obtained from the Proteins dataset

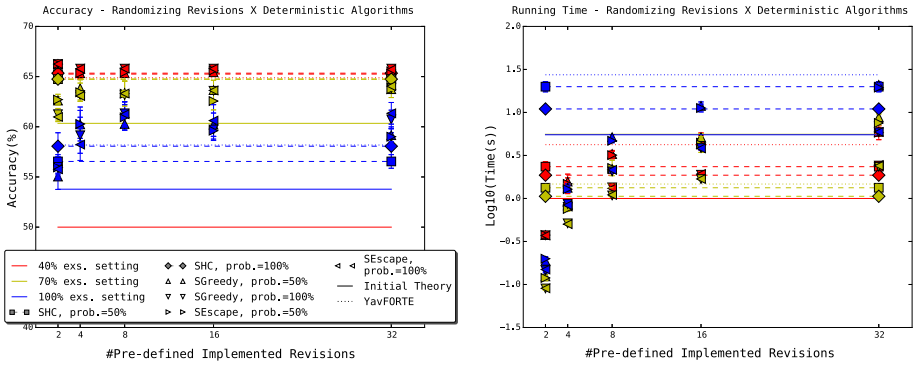

Fig. 16 Markers with the accuracy (*left*) and running time in logarithmic scale (*right*), obtained from the Metabolism dataset

This is possibly due to the requirement that the chosen revision has a score constrained by the grasp-like function, irrespective of the kind of move that the Stochastic Greedy algorithm follows. After all, in most cases we observed that this grasp-like function also demands an improvement in the current score, which makes both algorithms present similar accuracy results.

Thus, observe that the stochastic approaches reach the best results when the stopping criteria is set to 32 implemented revisions. The most evident exception is the Metabolism dataset, where the results are already good for the stochastic algorithms even with only 4 implemented revisions. In general, this is still less than the number of iterations of the deterministic approach. Once again, the probabilities seem to have little influence on the final results of accuracies.

Finally, the running time of the stochastic algorithms with the best stopping criteria parameters for the accuracies is always lower than the running time of the deterministic approach. Concisely, the closer results are in the Metabolism dataset, where the small size of the initial theories does not make the deterministic algorithm have a hard time.

Randomizing Literals Finally, we present the results of randomizing the literals to be added to/deleted from clauses in Fig. 17. First of all, the stochastic algorithm is always capable of improving the initial theories. Moreover, when comparing the accuracies achieved within all three settings of initial theories (40, 70 and 100) to the accuracy obtained by the learning from scratch with the whole set of examples (100), the revision also performs better. Furthermore, in Proteins dataset, the best accuracies of the stochastic version regarding the 5 runs, with both probabilities, are significantly better than the deterministic algorithm, as well as the average values of the 70 % setting. Proteins dataset has a preference for large clauses (Muggleton et al. 2010a) and the randomization of literals also tends to yield large clauses, as this algorithm converges slowly, sometimes even reaching the maximum number of literals in the clauses.

This also happens in other datasets, although less regularly, namely in one setting of the Metabolism dataset (100 % training set, 100$ of probability) and with Yeast Sensitivity best accuracies in three settings (40 % of the training set, 50 % of probability; 70 % of the training set, 100 % of probability; 100 % training set, 50$ of probability). In almost all the others cases, including the whole set of experiments with Pyrimidines, the accuracies of both the stochastic and deterministic versions do not present significant difference. The only exception is the Metabolism dataset, with the 40 % setting and 100 % of probability, where

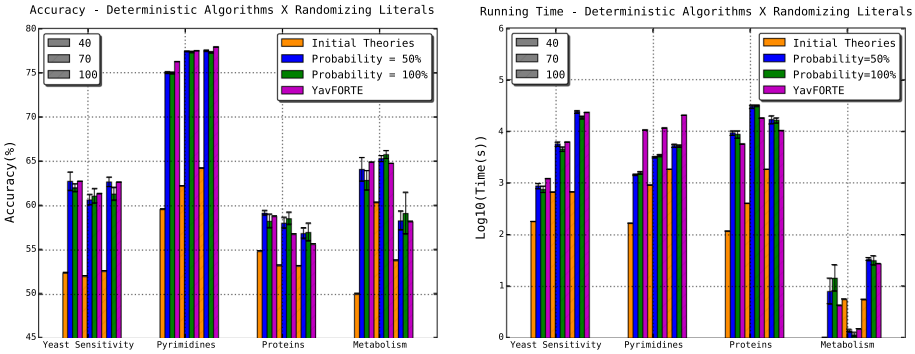

Fig. 17 Bar plot with the accuracies (*left*) and running time in logarithmic scale (*right*), obtained from all benchmark datasets. The results are obtained from the Algorithm that randomizes literals to be added to clauses

the stochastic version behaves worse. Notice that this is the setting where the initial theory has only the trivial clause, which may suggest that it is not a good idea to randomize literals when creating theories from scratch within a revision procedure.

However, the running time is not reduced as we would expect. Observe that the greatest reduction in time is in the Pyrimidines dataset, where the final accuracies are practically the same in both systems. Yeast Sensitivity, which is the dataset with the largest set of facts, has a running time behavior similar to Pyrimidines, except that the accuracies are higher or the same as in the deterministic version. The datasets that got highest accuracies also got the highest running time. Besides taking longer for the algorithm itself to converge, when the stochastic algorithm suggests a revision composed of large clauses, and this such revision is selected to be implemented, all the next coverage tests will also take more time to execute. In addition, the more literals there are in the clause, the larger the search space of next candidate literals is. This happens because more literals in a clause indicate more valid literals in the Bottom Clause according to the modes. Based on both the accuracies and running time results, it is advisable to use the stochastic randomization of literals in cases where large clause are better to represent the domain, since higher accuracies are probably to be achieved, with little cost in the running time.

Comparing all the algorithms and a combination of them Figures 18 and 19 compare the accuracy and running time of all the results presented before, together with the results of the revision process that combines three stochastic strategies, namely, randomizing revision points (choosing at most ten of them in a random walk), randomizing literals and randomizing revisions with the stochastic Hill-Climbing. We varied the probabilities of the random walks of these three strategies, considering 50 and 100 %, yielding in the end eight cases. As before, we present the results obtained from initial theories generated with 40, 70 and 100 % of the training set.

Concerning the results of the combination of the three strategies, the majority of them presented no significant difference in accuracies compared to the deterministic revision, while the running time is greatly reduced by one order or more of magnitude, sometimes even being faster than the ILP system. There are even cases where the accuracy is higher than in the deterministic approach (Metabolism dataset, 100 % of examples setting, for example), although not significantly better. The only dataset that escapes from this pattern is Pyrimidines, where a number of the combined results were significantly worse than the deterministic revision.

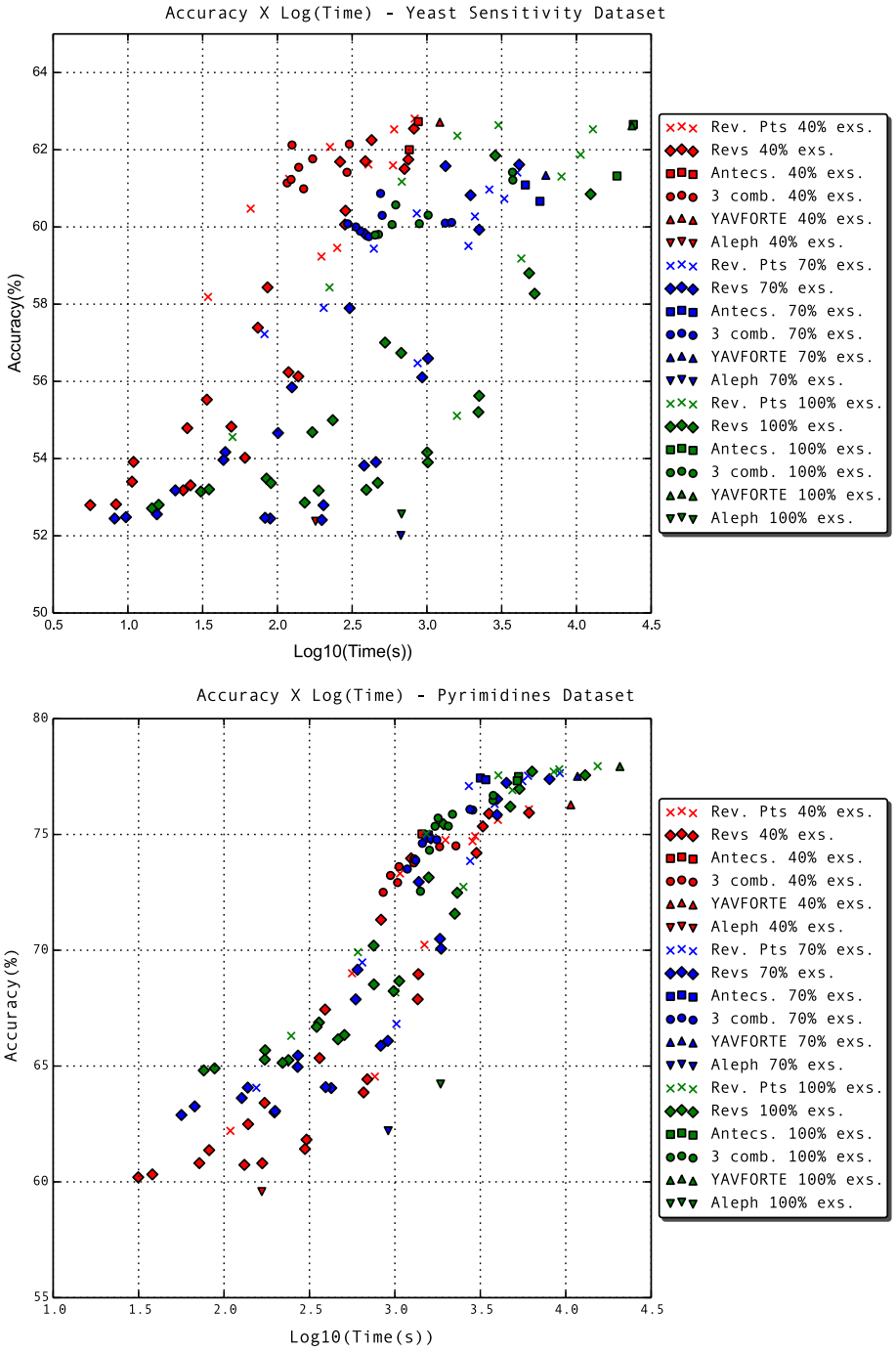

Fig. 18 Scatter plot comparing running time and accuracy of the deterministic approaches, the randomization with all strategies and a combination of the strategies for the Yeast Sensitivity (*top*) and Pyrimidines (*down*) datasets

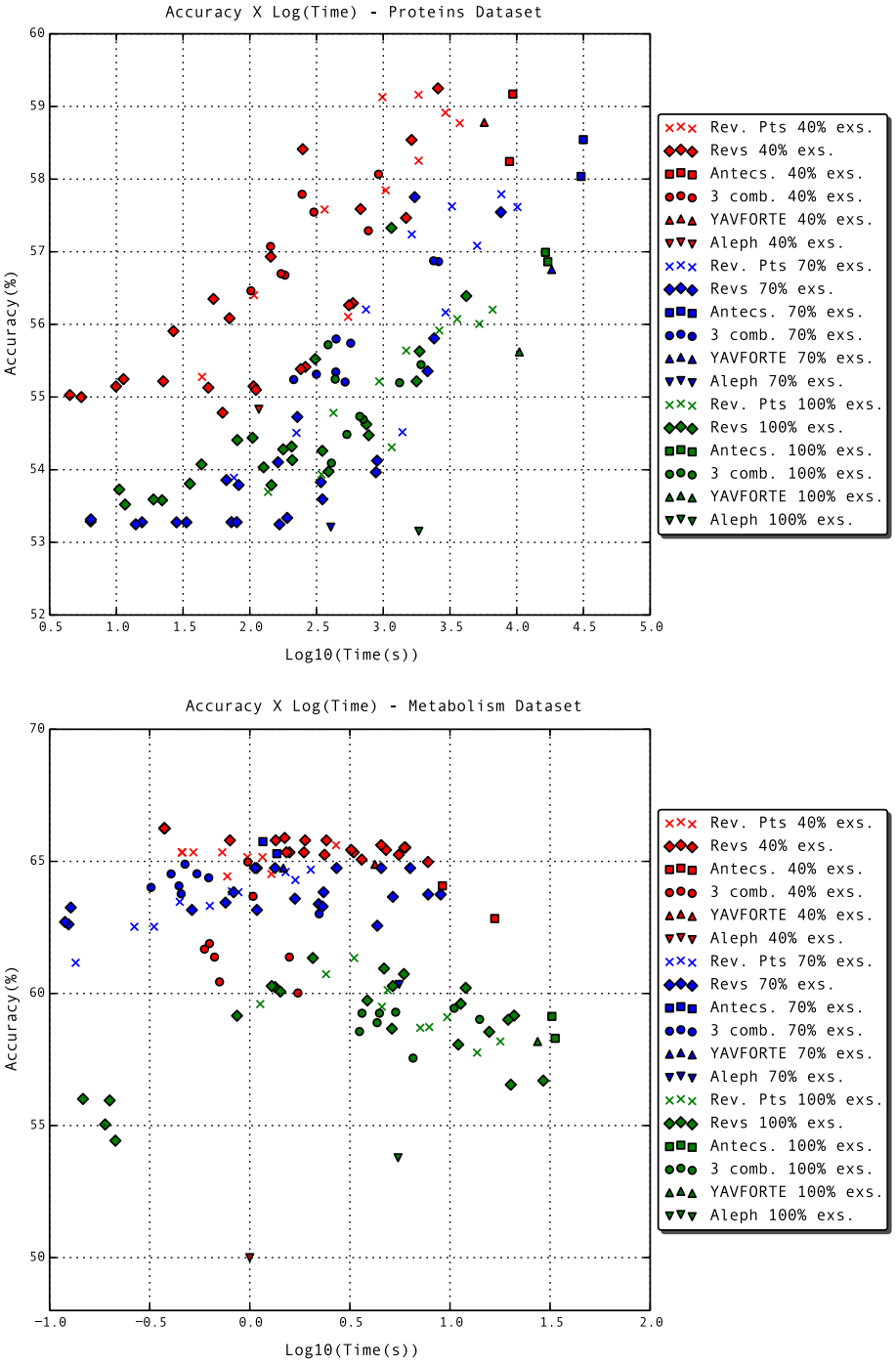

Fig. 19 Scatter plot comparing running time and accuracy of the deterministic approaches, the randomization with all strategies and a combination of the strategies for the Proteins (*top*) and Metabolism (*down*) datasets

Remember that this dataset has the largest initial theories, which in turn, in this case, also yielded a large number of revision points. Therefore, it seems that in this case it is not a good idea to reduce that much the search spaces altogether.

Once again, as also happened with the human-engineered theory, the best trade-off between accuracy and running time are achieved by either the combined approach or the randomization of revision points alone. Randomizing the revisions only performs a good job in a few cases where they are able to reduce the runtime without significantly decreasing the accuracy. However, in the Metabolism dataset, the majority of the results obtained from randomizing the revisions achieve competitive or better accuracies than the deterministic approach.

6 Conclusions

In this paper a set of stochastic local search algorithms were presented, for exploring the key search spaces of the revision process more efficiently. The algorithms abandon completeness in favor of finding good solutions in a reasonable time. They are based on random walks, so that the choice of pursuing a greedy or a stochastic move is made according to a probability parameter. Stochastic algorithms were implemented in YAVFORTE system in every key search of the revision process.

First, an SLS algorithm was built to avoid collecting all revision points from all misclassified examples. With a probability p, misclassified examples are randomized and a pre-defined number of revision points is generated. The search is alternated with greedy moves, since without the probability p all revision points found in the theory from each misclassified instance are collected. Through experimental results we found out that in a number of cases there is no need of employing greedy moves, since defining the probability parameter as 100 % already achieves good results: the revision time is greatly reduced, accuracies of the initial theories are greatly improved at the same time that they are not statistically significant different compared to the baseline revision system, when an appropriate number of revision points is selected. Thus, the performance of the stochastic component in the search for revision points is more influenced by the parameter defining the number of revision points that should be returned.

Second, three different stochastic components were included to decide which revision is going to be implemented, yielding a randomization of revision operators. In the results, we could see that the simplest strategy, which with a certain probability randomizes revisions and implements the first one improving the score, achieved the best overall results. The size of initial theories has a great influence on these results, as in the majority of the cases, the biggest reduction in running time and best improvement of accuracies are reached within the largest theories. On the other hand, the algorithms that accept possibly bad moves (bounded by a grasp-like evaluation function) usually achieve good results only when a large number of iterations is set as the stopping criteria. In our results, this number is the largest we set. Such results are also better following the size of the theories, but, in this case, as probably the number of iterations is not big enough, the best results are achieved with the setting with smallest theories.

Third, SLS components were included in the search for literals to be added to or removed from a clause. Stochastic search was included in both hill climbing and relational pathfinding algorithms for specializing clauses. In the first case, when the move is stochastic, literals from the Bottom Clause are randomized and the first literal found that improve the score is added to the clause. In relational pathfinding, paths created from the Bottom Clause and a positive

instance are randomized. In this work, we used both algorithms cooperating with each other, when relational pathfinding was applicable. We noticed from the empirical evaluation that, this approach is more likely to improve the accuracies, reaching higher values than the deterministic approach, than to reduce the runtime. Actually, in two datasets the running time was higher than the deterministic approach. The reasons for that are mainly due to the Bottom clause: either it is small or with few valid literals according to the modes, and, therefore, does not produce a large search space of literals, or it is large but in this case the use of stochastic move takes more iterations to converge than the original approach. Once again, it is better to always make random moves, instead of alternating them with greedy approaches.

To sum up, we showed in this paper that the revision process is able to improve the initial score of theories, in three scenarios: (1) with a human crafted initial theory, (2) with initial theories generated from training sets with reduced size and (3) with initial theories generated from the same set of examples provided to the revision system. Moreover, in several cases the stochastic algorithms were able to reduce the runtime while still improving the accuracy of the initial theories. Additionally, in some experiments the stochastic algorithms were able to achieve even higher accuracies than the deterministic approach. Putting our contribution together with the fast development of rich knowledge bases in areas such as biology (Muggleton 2005), suggests a strong case for theory revision.

There is a number of open directions for future work. First, we observed that the successful results obtained by YAVFORTE benefit from (1) the use of specialization and generalization operators, both at the clause and theory level; and (2) the use of stochastic search. We would like to investigate whether these techniques can be as effective on other ILP settings, such as learning from scratch or when using different theory revision algorithms. We believe that the rapid progress in stochastic search deserves further research in ILP.

Second, we would like to add to YAVFORTE rapid randomized restart strategies, already employed in ILP (Železný et al. 2006), that would avoid being stuck in unproductive refinements by restarting after a certain criteria from another random point.

Third, we would like to develop general mechanisms for obtaining initial theories from experts, supported by the recent progress in collective intelligence (Woolley et al. 2010). Also, some domains receive data in streams and we believe that they could benefit from an incremental revision process. Finally, we believe that it is straightforward to apply the stochastic local methods developed here when revising probabilistic logical models, starting, for example, from PFORTE (Paes et al. 2005) and BFORTE (Paes 2011) systems.

Acknowledgements Aline Paes and Gerson Zaverucha would like to thank the Brazilian Research Agency CNPq (483448/2013-3; 304399/2013-2). Vítor Santos Costa gratefully acknowledges Project "NanoSTIMA: Macro-to-Nano Human Sensing: Towards Integrated Multimodal Health Monitoring and Analytics/ NORTE-01-0145-FEDER-00001" financed by the North Portugal Regional Operational Programme (NORTE 2020), under the PORTUGAL 2020 Partnership Agreement, and through the European Regional Development Fund (ERDF). We would like to thank all the authors of previous work that made their systems and datasets publicly available. We also would like to thank the anonymous reviewers and the MLJ action editor for the profoundly valuable comments.

References

Adé, H., Malfait, B., & De Raedt, L. (1994). RUTH: An ILP theory revision system. In *8th international symposium on methodologies for intelligent systems (ISMIS-94), LNCS* (Vol. 869, pp. 336–345). Springer

Blockeel, H., & De Raedt, L. (1998). Top-down induction of first-order logical decision trees. *Artificial Intelligence*, *101*(1–2), 285–297.

Bratko, I. (1999). Refining complete hypotheses in ILP. In *Proceedings of the 9th inductive logic programming (ILP-99), LNAI* (Vol. 1634, pp. 44–55) Springer.

Buntine, W. (1991). Theory refinement on Bayesian networks. In *Proceedings of the 17th annual conference on uncertainty in artificial intelligence (UAI-91), San Mateo, CA* (pp. 52–60).
Cheng, J., Hatzis, C., Hyashi, H., Krogel, M. A., Morishita, S., Page, D., et al. (2002). KDD Cup 2001 report. *SIGKDD Explorations*, *3*(2), 47–64.
Chisholm, M., & Tadepalli, P. (2002). Learning decision rules by randomized iterative local search. In *Proceedings of the 19th international conference on machine learning (ICML-02)* (pp. 75–82).
De Raedt, L., & Bruynooghe, M. (1993). A theory of clausal discovery. In *Proceedings of the 13th international joint conference on artificial intelligence (IJCAI-93)* (pp 1058–1063).
Dietterich, T., Domingos, P., Getoor, L., Muggleton, S., & Tadepalli, P. (2008). Structured machine learning: The next ten years. *Machine Learning*, *73*, 3–23.
Duboc, A. L., Paes, A., & Zaverucha, G. (2009). Using the bottom clause and modes declarations on FOL theory revision from examples. *Machine Learning*, *76*(1), 73–107.
Dzeroski, S., & Bratko, I. (1992). Handling noise in inductive logic programming. In *Proceedings of the 2nd international workshop on inductive logic programming*.
Esposito, F., Semeraro, G., Fanizzi, N., & Ferilli, S. (2000). Multistrategy theory revision: Induction and abduction in INTHELEX. *Machine Learning*, *38*(1–2), 133–156.
Gama, J. (2010). *Knowledge discovery from data streams*. Boca Raton: CRC Press.
Gama, J., Medas, P., Castillo, G., & Rodrigues, P. (2004). Learning with drift detection. In *Advances in artificial intelligence-SBIA 2004* (pp. 286–295) Springer.
Garcez, A., & Zaverucha, G. (1999). The connectionist inductive learning and logic programming system. *Applied Intelligence*, *11*, 59–77.
Hirst, J. D., King, R. D., & Sternberg, M. J. E. (1994). Quantitative structure-activity relationships by neural networks and inductive logic programming. I. The inhibition of dihydrofolate reductase by pyrimidines. *Journal of Computer Aided Molecular Design*, *8*(4), 405–420.
Hoos, H. H., & Stützle, T. (2005). *Stochastic local search: Foundations and applications* (1st ed.). California: Elsevier.
Joshi, S., Ramakrishnan, G., & Srinivasan, A. (2008). Feature construction using theory-guided sampling and randomised search. In *Proceedings of the 18th international conference on ILP, LNAI* (Vol. 5194, pp. 140–157) Springer.
Kadupitige, S. R., Julia, K. C. L., Sellmeier, S. J., Catchpoole, D. R., Bain, M., & Gaeta, B. A. (2009). MINER: Exploratory analysis of gene interaction networks by machine learning from expression data. *BMC Genomics*, *10*(Suppl 3), S17.
King, R. D., Muggleton, S., & Sternberg, M. (1992). Drug design by machine learning: The use of inductive logic programming to model the structure-activity relationships of trimethoprim analogues binding to dihydrofolate reductase. *Proceedings of the National Academy of Sciences*, *89*(23), 11,322–11,326.
Kohavi, R. (1995). A study of cross-validation and bootstrap for accuracy estimation and model selection. In *Proceedings of the 14th international joint conference on artificial intelligence (IJCAI-95)* (pp 1137–1145).
Kowalski, R. A., & Kuehner, D. (1971). Linear resolution with selection function. *Artificial Intelligence*, *2*(3/4), 227–260.
Lloyd, J. (1987). *Foundations of logic programming* (2nd ed.). Berlin: Springer.
Muggleton, S. (1995). Inverse entailment and Progol. *New Generation Computing*, *13*(3&4), 245–286.
Muggleton, S. (2005). Machine learning for systems biology. In *Proceedings of the 15th international conference on inductive logic programming (ILP-05), lecture notes in computer science* (Vol. 3625, pp. 416–423) Springer.
Muggleton, S., & Bryant, C. H. (2000). Theory completion using inverse entailment. In *Proceedings of the 10th international conference on ILP, LNAI* (Vol. 1866, pp. 130–146) Springer.
Muggleton, S., & Tamaddoni-Nezhad, A. (2008). QG/GA: A stochastic search for Progol. *Machine Learning*, *70*(2–3), 121–133.
Muggleton, S., Paes, A., Costa, V. S., & Zaverucha, G. (2010a). Chess revision: Acquiring the rules of chess variants through FOL theory revision from examples. In *Inductive logic programming, 19th international conference, ILP 2009. Revised papers, LNCS* (Vol. 5989, pp. 123–130) Springer.
Muggleton, S., Santos, J. C. A., & Tamaddoni-Nezhad, A. (2010b). ProGolem: A system based on relative minimal generalisation. In *Proceedings of the 1th international conference on inductive logic programming (ILP-09), LNAI* (Vol. 5989, pp. 131–148) Springer.
Muggleton, S. H., King, R. D., & Sternberg, M. J. E. (1992). Protein secondary structure prediction using logic-based machine learning. *Protein Engineering*, *5*(7), 647–657.
Paes, A., Revoredo, K., Zaverucha, G., & Santos Costa, V. (2005). Probabilistic first-order theory revision from examples. In *Proceedings of the 15th international conference on inductive logic programming (ILP-05), LNAI* (Vol. 3625, pp. 295–311) Springer.

Paes, A., Železný, F., Zaverucha, G., Page, D., & Srinivasan, A. (2006). ILP through propositionalization and stochastic k-term DNF learning. In *Proceedings of the revised papers of 16th international conference on ILP (ILP-06), LNAI* (Vol. 4455, pp. 379–393) Springer.
Paes, A., Zaverucha, G., & Santos Costa, V. (2007). Revising first-order logic theories from examples through stochastic local search. In *Proceedings of the 17th international conference on ILP (ILP-07), LNAI* (Vol. 4894, pp. 200–210) Springer.
Paes, A. M. (2011). On the Effective revision of (Bayesian) logic programs from examples. Ph.D. thesis.
Quinlan, J. (1990). Learning logical definitions from relations. *Machine Learning*, *5*, 239–266.
Ramachandran, S., & Mooney, R. (1998). Theory refinement of Bayesian networks with hidden variables. In *Proceedings of the 15th international conference on machine learning (ICML-98)* (pp. 454–462).
Richards, B. L., & Mooney, R. J. (1992). Learning relations by pathfinding. In *Proceedings of the 10th annual national conference on artificial intelligence (AAAI-92)* (pp. 50–55).
Richards, B. L., & Mooney, R. J. (1995). Automated refinement of first-order Horn-clause domain theories. *Machine Learning*, *19*(2), 95–131.
Richardson, M., & Domingos, P. (2006). Markov logic networks. *Machine Learning*, *62*(1–2), 107–136.
Rückert, U., & Kramer, S. (2003). Stochastic local search in k-term DNF learning. In *Proceedings of the 20th international conference on machine learning (ICML-03)* (pp. 648–655).
Rückert, U., & Kramer, S. (2004). Towards tight bounds for rule learning. In *Proceedings of the 21st international conference on machine learning (ICML-04), ACM* (Vol. 69).
Russell, S., & Norvig, P. (2010). *Artificial intelligence: A modern approach* (3rd ed.). Englewook Cliffs, NJ: Prentice-Hall.
Santos Costa, V., Damas, L., & Rocha, R. (2012). The yap prolog system. *Theory and Practice of Logic Programming*, *12*(Special Issue 1–2), 5–34.
Selman, B., Levesque, H., & Mitchell, D. (1992). A new method for solving hard satisfiability problems. In *Proceedings of the 10th annual national conference on artificial intelligence (AAAI-92)* (pp. 440–446).
Selman, B., Kautz, H. A., & Cohen, B. (1996). Local search strategies for satisfiability testing. Cliques, coloring, and satisfiability: Second DIMACS implementation challenge, October 11–13, 1993. In *DIMACS series in discrete mathematics and theoretical computer science* (Vol. 26, pp. 521–532).
Serrurier, M., & Prade, H. (2008). Improving inductive logic programming by using simulated annealing. *Information Sciences*, *178*(6), 1423–1441.
Shapiro, E .Y. (1981). The model inference system. In *Proceedings of the 7th international joint conference on artificial intelligence (IJCAI-81)* (p. 1064) William Kaufmann.
Specia, L., Srinivasan, A., Joshi, S., Ramakrishnan, G., & Nunes, M. D. G. V. (2009). An investigation into feature construction to assist word sense disambiguation. *Machine Learning*, *76*(1), 109–136.
Srinivasan, A. (2000). A study of two probabilistic methods for searching large spaces with ILP. Tech. Rep. PRG-TR-16-00, Oxford University Computing Laboratory, Oxford.
Srinivasan, A. (2001). The Aleph Manual. http://web.comlab.ox.ac.uk/oucl/research/areas/machlearn/Aleph/aleph.html.
Tamaddoni-Nezhad, A., & Muggleton, S. (2000). Searching the subsumption lattice by a genetic algorithm. In *Proceedings of the 10th international conference on ILP (ILP-00), LNAI* (Vol. 1866, pp. 243–252) Springer.
Towell, G., & Shavlik, J. (1994). Knowledge-based artificial neural networks. *Artificial Intelligence*, *70*(1–2), 119–165.
Widmer, G., & Kubat, M. (1996). Learning in the presence of concept drift and hidden contexts. *Machine Learning*, *23*(1), 69–101.
Wogulis, J., & Pazzani, M. (1993). A methodology for evaluationg theory revision systems: Results with Audrey II. In *Proceedings of the 13th international joint conference on artificial intelligence (IJCAI-93)* (pp. 1128–1134).
Woolley, A. W., Chabris, C. F., Pentland, A., Hashmi, N., & Malone, T. W. (2010). Evidence for a collective intelligence factor in the performance of human groups. *Science*, *330*(6004), 686–688.
Wrobel, S. (1994). Concept formation during interactive theory revision. *Machine Learning*, *14*(1), 169–191.
Wrobel, S. (1996). First-order theory refinement. In L. De Raedt (Ed.), *Advances in inductive logic programming* (pp. 14–33). Netherlands: IOS Press.
Železný, F., Srinivasan, A., & Page, D. (2002). Lattice-search runtime distributions may be heavy-tailed. In *Proceedings of the twelth international conference on inductive logic programming (ILP-02), LNAI* (Vol. 2583, pp. 341–358) Springer.
Železný, F., Srinivasan, A., & Page, D. (2006). Randomised restarted search in ILP. *Machine Learning*, *64*(1–3), 183–208.