Extending Probabilistic Theory Revision to First Order Theories

Bryan Singer
singer@dollar.biz.uiowa.edu

Alberto Maria Segre
segre@cs.uiowa.edu

Department of Management Sciences
The University of Iowa
Iowa City, IA 52242

1. Introduction

The problem of properly refining domain-specific knowledge obtained from human experts is a central problem in expert systems design and development. Typically, domain knowledge originally elicited from human experts tends not to be entirely accurate, or, if more than one expert is involved, consistent. The *theory revision problem* is concerned with how best to go about revising a domain theory on the basis of a collection of examples, some of which expose inaccuracies in the original domain theory. Note that there may be many possible revisions that properly account for all of the observed examples; our goal here is to find a revised domain theory that is both consistent with the examples and as faithful as possible to the expert's original domain theory.

This paper describes our work on *Probabilistic Theory Revision* (PTR), one approach to the theory revision problem. We begin with a review of our work on the PTR0 algorithm for propositional theory revision. We then show how this propositional solution has been extended to create the PTR1 algorithm, a probabilistic theory revision algorithm suitable for relational theories, an important subset of full first-order logic. We conclude with some remarks on remaining problems and future research directions.

2. PTR0: Probabilistic Theory Revision for Propositional Theories

In previous work, we have developed a novel theory revision system suitable for propositional definite clause theories. Our PTR0 algorithm demonstrates many advantages over other competing approaches to the theory revision problem. First, it is capable of exploiting additional information provided by the domain expert reflecting his or her initial confidence in each domain theory component. Second, it honors the principle of representational transparency by retaining intact the structure of the domain theory during the revision process, facilitating inspection of the changes it performs by the human expert. Third, PTR0 is efficient, relying on an incremental algorithm that is of low complexity with respect to the number of observed examples in the most time-critical portion of the revision process. Fourth, the nature of the PTR0 algorithm makes it uniquely suited to parallel implementation on SIMD (single instruction multiple dataset) parallel hardware. Fifth, PTR0 exhibits the desirable formal properties of termination and convergence by virtue of its underlying probabilistic framework.

We begin by defining some necessary terminology, such as an exact definition of domain theory. Next, we introduce the notion of a weighted domain theory and describe the PTR0 algorithm for revising weighted domain theories.

2.1. Preliminaries

A *propositional domain theory*, denoted Γ, is a set of clauses c_i: $H_i \leftarrow B_i$ where c_i is a clause label, H_i is a proposition (called the *head* of c_i) and B_i is a set of positive and negative literals (called the *body* of c_i).[1] It may be convenient to think of this as a propositional logic program without facts. As usual, the clause c_i: $H_i \leftarrow B_i$ represents the assertion that the proposition H_i is implied by the conjunction of literals in B_i. The domain theory is simply the conjunction of its clauses.

A proposition that does not appear in the head of any clause is said to be *observable*. Propositions that do not appear in the body of any clause are called *roots*. A *fact* is an assertion that some observable proposition is true. An *example*, E, is a set of facts. All observable propositions whose truth is not

[1] We assume familiarity with basic logic terms such as *clause*, *literal*, and *proposition*. Standard definitions can be found in [Lloyd87].

asserted in E are assumed to be false in E; in other words, we make a closed world assumption.

When combined with an inference procedure (for simplicity, we assume the SLD resolution model of Prolog), the domain theory constitutes an expert system. The expert system is capable of drawing conclusions based on a set of facts describing some real-world situation, where the conclusions of interest typically correspond to roots of the domain theory. For example, consider the following artificial domain theory:

$$c_1\colon r \leftarrow a \wedge b$$
$$c_2\colon r \leftarrow c \wedge d$$

where r is a root proposition and $\{a, b, c, d\}$ constitute the set of observable propositions. If we consider example $E_1 = \{a, c, d\}$, then, we see that r is in fact provable. However, for example $E_2 = \{a, c\}$, r is not provable.

Now we can devise a more formal statement of the general theory revision problem. Let Γ be a domain theory with roots $r_1, .., r_n$. For an example E, we define the vector $\Gamma(E) = \langle \Gamma_1(E), \ldots, \Gamma_n(E) \rangle$ where $\Gamma_i(E) = 1$ if $E + \Gamma \vdash r_i$ (using resolution) and $\Gamma_i(E) = 0$ if $E + \Gamma \not\vdash r_i$. Intuitively, $\Gamma(E)$ tells us which of the conclusions $r_i, \ldots, r_n$ will be made by the expert system when given the initial set of facts E.

Let the *target domain theory* Θ be some domain theory that correctly models the domain of interest. In other words, Θ represents the correct expert system domain theory. An ordered pair $\langle E, \Theta(E) \rangle$ is called an *exemplar* of the domain: if $\Theta_i(E) = 1$ then the exemplar is said to be an *IN exemplar* of r_i, while if $\Theta_i(E) = 0$ then the exemplar is said to be an *OUT exemplar* of r_i. Let Γ be some possibly incorrect theory for a domain which is in turn correctly modeled by the target theory Θ (*i.e.*, Γ represents the initial theory provided by the expert). Any inaccuracies in Γ will be reflected by exemplars for which $\Gamma_i(E) \neq \Theta_i(E)$. Such exemplars are said to be *misclassified* by Γ. Thus a *misclassified IN exemplar* of r_i, or *false negative*, will have $\Theta_i(E) = 1$ but $\Gamma_i(E) = 0$, while a *misclassified OUT exemplar* of r_i, or *false positive*, will have $\Theta_i(E) = 0$ but $\Gamma_i(E) = 1$.

A *revision operator* applied to a domain theory Γ transforms the domain theory in some well-specified fashion. There are only four possible revision operators available for use with propositional domain theories of the form just described:

add a literal to an existing clause,
delete an existing clause,
add a new clause, and
delete a literal from an existing clause.

For negation-free domain theories, the first two operations result in *specializing* Γ, since they may allow some (presumably misclassified) IN exemplars to become OUT exemplars. The latter two operations result in *generalizing* Γ, since they may allow some (presumably misclassified) OUT exemplars to become IN exemplars. Note that if a domain theory contains negations, then, depending on the location of the negations, the first two operators may in fact generalize Γ, while the latter two operations may in fact specialize Γ.

We say a set of revisions to Γ is *adequate* if, after the revision operators are applied, all known exemplars are correctly classified by the revised domain theory Γ'. Note that we are not implying that $\Gamma' = \Theta$, but rather that for all known exemplars E_i, $\Gamma'(E_i) = \Theta(E_i)$. There may be more than one adequate revision set; the goal of any theory revision system, then, is to find the "best" adequate revision set for Γ given a set of exemplars according to some fixed sense of goodness. Most previous work on propositional theory revision relies on a relatively narrow notion of which adequate revision set is best: it is simply the smallest such set. More precisely, Γ' is the domain theory syntactically closest to the original domain theory Γ that correctly classifies all known exemplars. In contrast, we define a probabilistic notion of a best adequate revision set based on the concept of weighted domain theories.

A domain theory, Γ, is said to be *weighted* if each element e (the collective term for clauses and literals) of Γ is assigned some value $0 \le w(e) \le 1$. These weights reflect the expert's *a priori* confidence in the elements of a given domain theory; they serve to lift the assumption — implicit in domain theory revision systems that minimize syntactic change — that each domain theory element is of equal importance.[2] More formally, the weight assigned to an edge is taken as the expert's degree of confidence that the corresponding domain theory element will remain a part of the target theory Θ.

Given a weighted domain theory, how can we know which adequate revision set should be preferred? Since Γ is weighted the answer to this question is clear: the most probable set of adequate revisions should be preferred [Koppel94]. While the notion of most probable revision set can be defined quite precisely, such formal arguments are beyond the scope of this paper. Instead, we summarize intuitively the operation of PTR0 as follows:

> *PTR0 is a greedy algorithm that, given a set of exemplars, finds an adequate set of revisions approximating the most probable adequate set of revisions for an initial weighted domain theory.*

Roughly speaking, PTR0 considers each known exemplar one at a time and efficiently updates the weights associated with each element of Γ. Somewhat more precisely, if a literal or clause contributes to the correct classification of an exemplar, its weight increases; if it contributes to the misclassification of an exemplar, its weight decreases. When the weight associated with an element of Γ drops below a prespecified revision threshold σ, it is an indication that this element of Γ should be revised. PTR0 selects the appropriate revision operation (either deletion of the element in question or addition of a clause or literal) and revises the indicated element. PTR0 continues to process exemplars — possibly cycling through the set of known exemplars more than once — until all of the exemplars provided are classified correctly by the revised domain theory.

[2] Ideally, the expert would assign individual weights to each element in the domain theory. Clearly, this would be impractical for domain theories of non-trivial size; thus, most (or even all) of the weights may be assigned default values according to some prespecified assignment strategy. While PTR0 has been shown to be relatively robust even in the face of poorly selected weights, experiments reported elsewhere [Koppel94] show PTR0 is effective even if weights are initially assigned default values, experiments reported elsewhere show that sensible assignment of at least some initial weights by a domain expert can reduce the time required for convergence.

2.2. Updating Weights

The core of the PTR0 algorithm is the efficient updating of weights given a single exemplar. Our procedure relies on a special data structure which is used to represent the domain theory Γ. A *DT graph* for domain theory Γ, denoted Δ_Γ, is a collection of nodes and edges such that each unique clause and literal in Γ has a corresponding node in Δ_Γ. Edges in Δ_Γ are constructed according to three relations. First, a *literal edge* connects each node in Δ_Γ representing a clause c_i in Γ to each node in Δ_Γ representing corresponding positive or negative literals in B_i, the body of c_i. Second, a *clause edge* connects each node representing a head literal H_i in Δ_Γ to the node in Δ_Γ representing the corresponding clause c_i in Γ. Finally, a *negation edge* connects each node representing a negative literal in Δ_Γ to the node representing its corresponding positive literal in Δ_Γ. Given two edges e_1 and e_2 in Δ_Γ, we can say that e_1 is a *parent edge* of e_2 if the destination node of e_1 is the same as the source node of e_2, that is, $e_1 \in P(e_2)$ where $P(e)$ denotes the parent edges of e. Similarly, if $e_1 \in P(e_2)$ then e_2 is called a *child edge* of e_1. We use $C(e_1)$ to denote the set of child edges of e_1.

We extend Δ_Γ by adding extra *root pseudo edges* (directed edges, denoted e_{r_i}, from some imaginary supernode to each root node r_i). In this way, every root of Γ also has exactly one parent edge. We can now associate the weights for elements of Γ directly with their parent edges in Δ_Γ (weights for negative literals are associated with the corresponding literal edge, while negation edges always receive unit weight). Assigning weights to edges as opposed to nodes in Δ_Γ has the advantage of allowing distinct weights to be associated with each occurrence of a clause or literal, since an edge will have more than one parent edge in precisely those situations where the same clause or literal appears more than once in the domain theory.

The update algorithm takes as input an exemplar and a weighted DT graph Δ_Γ and updates the weights associated with the edges in Δ_Γ. The updating algorithm requires the use of two additional values, $u(e)$ and $v(e)$, associated with each edge e in Δ_Γ to store intermediate results. The algorithm proceeds in two stages. In the first stage, starting from the leaf nodes of Δ_Γ, we propagate information corresponding to the observable propositions in the exemplar up through Δ_Γ, computing a value for $u(e)$ for every edge in Δ_Γ. Intuitively, we think of $u(e)$ as the "proof flow" through the edge e and $u(r_i)$ as the approximate probability that the root r_i for a given exemplar (set of argument bindings) is true. More precisely, for every edge e_{obs} in Δ_Γ corresponding to an observable proposition in Γ:

$$u(e_{obs}) = \begin{cases} 1 & \textit{if } e_{obs} \in E \\ 1 - w(e_{obs}) & \textit{otherwise} \end{cases} \tag{1}$$

Now we proceed bottom up through Δ_Γ, computing $u(e)$ values for the remaining edges:

$$u(e) = \begin{cases} 1 - w(e) & \textit{if } C(e) = \{\} \\ 1 - w(e) \cdot \prod\limits_{e_i \in C(e)} u(e_i) & \textit{otherwise} \end{cases} \tag{2}$$

At the end of this first stage, then, we will have computed a value $u(e_{r_i})$ for every root pseudo edge e_{r_i} in Δ_Γ.

In the second stage of the updating algorithm, we propagate the difference between each computed root pseudo edge value $u(e_{r_i})$ and the desired value $\Theta_i(E)$ (essentially a classification error term) top-down through Δ_Γ in a process similar to backpropagation in neural networks. As we proceed, we compute values of $v(e)$ and new values for $w(e)$ for every edge e in Δ_Γ. We begin by setting each root pseudo edge's $v(e_{r_i})$ value to reflect the correct classification of the exemplar:

$$v(e_{r_i}) = \begin{cases} 1 - \varepsilon & \textit{if } \Theta_i(E) = 1 \\ \varepsilon & \textit{if } \Theta_i(E) = 0 \end{cases} \tag{3}$$

where ε is some small constant $0 < \varepsilon < 1$. Now we proceed top down through Δ_Γ, computing $v(e)$ for each edge in Δ_Γ.

$$v(e) = 1 - (1 - u(e)) \cdot \frac{v(p)}{u(p)} \tag{4}$$

where p is that element of $P(e)$ for which $\left| 1 - \dfrac{\max[v(e), u(e)]}{\min[v(e), u(e)]} \right|$ is greatest.[3] If there is only one parent edge, p will naturally correspond to that edge. In the event that there are multiple parents, this method of selecting p will ensure the current exemplar has the maximum possible effect on the weights by propagating the greatest change for each edge corresponding to multiply-occurring elements of the domain theory.

We next compute new values for $w(e)$ using the previous value of $w(e)$ and the values of $v(e)$ and $u(e)$ just computed:

$$w(e) = 1 - (1 - w(e)) \cdot \frac{v(e)}{u(e)} \tag{5}$$

As before, we have shown elsewhere [Koppel94] that, in certain restrictive contexts, each newly computed value for $w(e)$ corresponds to the conditional probability that the domain theory element corresponding to edge e need not be revised given the classification of the exemplar.

This procedure for updating weights has several interesting characteristics. First, for any edge e, if $w(e) = 0$ or $w(e) = 1$ then PTR0 will never alter these values. Thus a domain theory element about which there is complete confidence will never change. Second, if $u(e) = 1$ or $u(e) = 0$ then no changes will be made to the weights of any edge in the subtree rooted at e. In particular, an exemplar with $u(e_{r_i}) = 0$ or $u(e_{r_i}) = 1$ will never alter any weights in Δ_Γ. Third, the computational complexity of processing a single exemplar is linear in the number of nodes in Δ_Γ. Thus the update algorithm is quite efficient when compared to many previously proposed revision techniques which rely on enumerating all proofs for a root. Finally, the computation required to update a weight (Equations 2, 4, and 5) is identical for every edge of Δ_Γ regardless of edge type. Thus, PTR0 is well suited for mapping onto a fine-grained SIMD

---

[3] Note that it is possible for $u(p)$ to be zero if $w(p) = 1$ and for all child edges $e \in C(p)$, $u(e) = 1$. This includes the case where edge p has no children. In these situations, we define $\dfrac{v(p)}{u(p)} = 0$.

(single instruction multiple dataset) parallel machine.

2.3. Applying a Revision Operator

Once an element has been selected for revision (the *culprit*), we must decide which of the revision operators should be applied to it. For this problem, PTR0 uses a procedure that is almost identical to that of the EITHER system [Ourston91, Ourston90].

Recall there are two types of revisions possible; either Γ is specialized, reducing the obtainable proofs, or Γ is generalized, augmenting the obtainable proofs. How do we decide whether Γ requires specialization or generalization? Consider, for example, a culprit clause c_j in need of revision. If we assume a negation-free domain theory, $w(e_{c_j})$ can only be reduced by the presence of undesirable proofs for OUT exemplars. Thus c_j must be specialized in order to restrict the proofs obtained. Similarly, a literal l_k in need of revision should be generalized, since $w(e_{l_k})$ is only reduced by the absence of desired proofs for IN exemplars. Similar arguments hold in the presence of negation.

While the choice of generalization versus specialization should be clear, it is somewhat more difficult to decide exactly which of two specialization or two generalization operators should be used. Recall there are a total of four revision operators for propositional theories; two specialization operators (adding a literal or deleting a clause) and two generalization operators (adding a clause or deleting a literal). While deleting clauses or literals will always correct a portion of the misclassified exemplars, it may cause other — previously correctly classified — exemplars to become misclassified. In the limit, a deletion-only revision strategy can only be successful if the original domain theory Γ is a supertheory of the target theory Θ (*i.e.*, $\Delta_\Gamma \supseteq \Delta_\Theta$).

A much less drastic alternative to deleting a selected clause is to specialize the clause by adding new literals to its body. The literals added should be chosen so that they prevent those OUT exemplars that have been exploiting the clause to achieve an undesired proof. On the other hand, the added literals should not inadvertently prevent proofs of IN exemplars that relied on this particular clause. A dual argument holds for selected literals; instead of deleting a literal, we can generalize by adding new clauses with the selected literal as the head in order to produce alternative proofs of the literal under appropriate conditions. As before, we must ensure that the added clauses do not inadvertently facilitate production of undesirable proofs of previously OUT exemplars.

We can rely on standard inductive inference techniques such as the ID3 algorithm [Quinlan86] to provide clauses or literals that will correct the problem without compromising desired proofs.[4] Suppose, for example, that culprit clause c_j has been selected for revision in a negation-free theory Γ. We partition the entire set of exemplars (not just those seen so far) into two subsets N and D corresponding to exemplars for which the clause is needed and for which the clause is destructive. The N subset consists of those exemplars where c_j contributes to a desired proof or where c_j prevents an undesired proof. Likewise, the D subset consists of those exemplars where c_j prevents a desired proof or where c_j contributes to an undesired proof. Note that $N \cup D$ may not comprise the entire set of exemplars, as many of the exemplars may not involve c_j at all. We call $N \cup D$ the set of *relevant* exemplars.

To determine the relevance of an exemplar, we perform a bottom-up on the exemplar once with the culprit edge weighted one and once with the culprit edge removed from the DT graph, each time noting the proof flow at the exemplar's root. Intuitively, an exemplar is relevant if these proof flows are significantly different. More formally, we say an IN exemplar is needed if the ratio of *u_with* over *u_without* is greater than two where *u_with* is the proof flow with the culprit edge weighted one and *u_without* is the proof flow with the culprit edge removed. An IN exemplar is destructive if the ratio of $1 -$ *u_with* over $1 -$ *u_without* is greater than two. Intuitively, we subtract the proof flows from one when

[4] ID3 is classic propositional inductive learning algorithm that constructs a decision tree from exemplar data. The decision tree produced by ID3 is easily reformulated as an expression in disjunctive normal form (DNF).

testing for an IN exemplar being in the destructive set because here we are wanting the ratio of how *un*provable the exemplar's root is. In a similar manner, an OUT exemplar is needed if the ratio of $1 - u\_with$ over $1 - u\_without$ is greater than two and destructive if the ratio of *u_with* over *u_without* is greater than two.

At this point, if the N set is empty, we simply delete the edge corresponding to c_j from Δ_Γ. We justify this deletion by noting that no exemplars require c_j, so deletion will not compromise the performance of the resulting theory. On the other hand, if N is not empty, we apply the inductive algorithm to produce a disjunctive normal form (DNF) logical expression constructed from observable propositions that is true for each exemplar in N but no exemplar in D. We reformulate this DNF expression as a conjunction of n clauses by taking a single new literal l_i as the head of each clause and using each conjunct in the DNF expression as the body of one of the n clauses. The clauses are converted into DT graph form and sutured into Δ_Γ via a new clause edge from c_j to l_i. This newly added substructure serves to filter out exemplars in D which might otherwise produce an unwanted proof via c_j, in effect specializing Γ.[5]

An analogous construction applies when attempting to generalize Γ by adding a clause to a culprit literal. Once again, if the N set is empty, removing the selected literal will not facilitate undesired proofs, so it may be deleted. Otherwise, the DNF expression produced by the inductive algorithm from N and D is transformed into a collection of clauses whose head literal corresponds to the culprit literal. By suturing new structure into Δ_Γ rooted at the culprit literal, Γ is generalized by providing additional proofs of the culprit literal.

[5] In the special case that the DNF expression consists of a single conjunct in turn composed of a single literal, the lone literal can be sutured directly into the DT graph as a new child of c_j.

When a new structure is sutured into Δ_Γ, we must adjust the weight of the culprit edge and assign weights to the edges in the new subcomponent. Unlike the original domain theory, the new substructure is an artifact of the inductive algorithm used and the current relevant exemplar set. For this reason, it is almost certainly inadvisable to try to revise it as new exemplars are encountered.[6] Instead, we would prefer that this new structure be removed as a unit and replaced with a more appropriate construct should the need arise. To ensure replacement instead of revision, we assign unit weights to all edges of the substructure, and we assign a default weight λ to both the substructure root edge that connects the new component to the existing Δ_Γ and the original culprit edge. Since the internal edges of the new structure have weights equal to 1, they will never be revised.[7]

2.4. Termination and Convergence

PTR0 continues processing exemplars, updating weights, and performing revisions whenever a weight falls below the revision threshold σ. When all exemplars have been processed, PTR0 extracts the revised version of the domain theory from Δ_Γ, ignoring all edges with weights equal to zero. The new domain theory is checked against all known exemplars, and, if they are all correctly classified, PTR0 terminates. If there are still some misclassified exemplars, PTR0 reprocesses these in random order, performing additional revisions and checking once again for termination against all known exemplars.[8] If at any point all weights in Δ_Γ become 1, PTR0 terminates since no future revisions are possible. Note that since, in principle, PTR0 may perform many cycles through the same set of exemplars, we would like

[6] Of course, the desirable representational transparency property does not apply to new substructure constructed by a traditional inductive learning algorithm as there is no corresponding original knowledge.

[7] In the special case described in Footnote 5, the new literal l would be given initial weight λ instead.

[8] This procedure relies the assumption that exemplars are independent and can therefore be considered equally effectively in any order. This same assumption is also commonly made by neural network systems. While this assumption is clearly not valid for most applications, empirical tests suggest that, although time to convergence may be affected, relaxing the assumption does not severely hamper the effectiveness of the algorithm.

some assurance that two important criteria will be met.

The first criteria we wish to guarantee is *termination*. Termination means that PTR0 will at some point stop processing exemplars and halt. We can guarantee termination by setting the revision threshold σ and the default substructure weight λ to some initial value between 0 and 1 (*e.g.*, 0.2) and resetting these values after each cycle as follows:

$$\sigma = \min(\sigma + \delta_\sigma, 1) \tag{6}$$

$$\lambda = \min(\lambda + \delta_\lambda, 1) \tag{7}$$

where revision threshold and new substructure threshold increments δ_σ and δ_λ are small constants. It should be clear that these conditions assure PTR0's termination: at some point, the revision threshold σ will be 1, causing every edge with weight smaller than 1 to be deleted or corrected via the addition of new substructure. Since λ also approaches 1, at some point all remaining edges are guaranteed to have a weight of 1. Since no further revision can take place at this point, PTR0 is forced to terminate.

The second criteria, *convergence*, states that, at termination, the resulting domain theory will correctly classify all known exemplars. Guaranteeing convergence is somewhat more difficult. We have shown elsewhere that, under certain restrictions[9], convergence is guaranteed by PTR0 using the termination conditions just described: since the proof is beyond the scope of this paper, the interested reader is referred to [Feldman93a].

---

[9] These assumptions include the independence of domain theory elements as well as certain structural restrictions on Δ_Γ. In particular, Δ_Γ must not only be acyclic, but only nodes corresponding to observable propositions are allowed to have more than one parent in Δ_Γ. While these assumptions are clearly not met in real applications, empirical tests suggest that relaxing these assumptions does not severely hamper the effectiveness of the algorithm. The version of PTR0 described in this paper relies on relaxing these assumptions as well as making a certain number of approximations in the interest of computational efficiency.

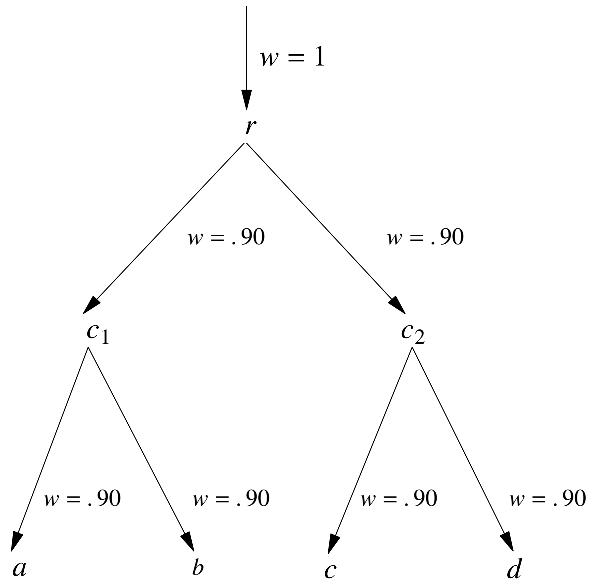

Figure 1: DT graph for sample domain theory. All edges have initially been assigned default weights $w(e) = .90$ with the exception of e_r which has been assigned $w(e_r) = 1$.

2.5. An Example

An example should help make this entire process clear. Figure 1 shows the initial DT graph Δ_Γ for the sample negation-free domain theory Γ described earlier. For this example, we have assigned by default $w(e) = .90$ for all edges e in Δ_Γ except the root edge, which is assigned $w(e_r) = 1$ (we ensure the root edge itself is not revised, since we don't wish to risk reducing our example to an invocation of ID3). Assume we are operating with parameters $\sigma = .20$, $\varepsilon = .10$, and $\lambda = .90$. We are given three exemplars, $E_1 = \langle \{a, c, e\}, 1 \rangle$, $E_2 = \langle \{a, b, c\}, 0 \rangle$, and $E_3 = \langle \{a, d, e\}, 1 \rangle$.

We first update Δ_Γ according to the exemplar $E_1 = \langle \{a, c, e\}, 1 \rangle$. E_1 is a misclassified IN exemplar, since $\Gamma(\{a, c, e\}) = 0$ while $\Theta(\{a, c, e\}) = 1$. Figure 2 shows the values computed after each stage of the update procedure. On the left, we see Δ_Γ after the bottom-up stage. First, the leaves are labeled according to their presence or absence in the exemplar (Equation 1). Next, $u(e)$ values are computed for the rest of the tree bottom up according to Equation 2. On the right, we see Δ_Γ after the top-down stage is complete. Since $\Theta(\{a, c, e\}) = 1$, the value of $v(e_{r_i})$ is set initially to $1 - \varepsilon$ (Equation 3). Values for $v(e)$ and updated values for $w(e)$ are computed for the remaining edges of Δ_Γ according to Equations 4 and 5.

Note that $w(b)$ and $w(d)$ have decreased, since they each censored a (desired) proof of query r. Since they are symmetric with respect to E_1, both weights are equally diminished. Similarly, $w(c_1)$ and $w(c_2)$ have been equally increased, given that each contributed identically to a partial proof of r (at this point, there is no reason to prefer one partial proof over the other).

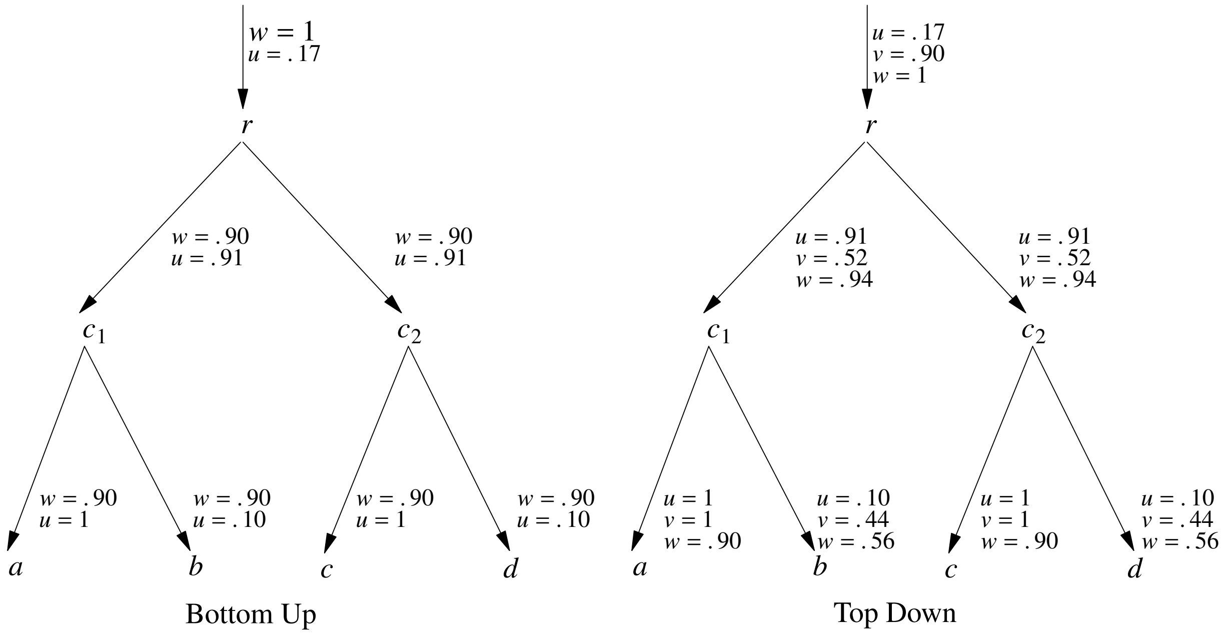

Figure 2: DT graph for sample domain theory during the updating of weights according to exemplar $E_1 = \langle \{a, c, e\}, 1 \rangle$.

Figure 3 shows the state of Δ_Γ during the processing of $E_2 = \langle \{a, b, c\}, 0 \rangle$. E_2 is a misclassified OUT exemplar, since $\Gamma(\{a, b, c\}) = 1$ while $\Theta(\{a, b, c\}) = 0$. After updating Δ_Γ, we note that $w(d)$, which had decreased in the course of processing E_1, has now increased since it serves to censor an undesired proof of r, while $w(b)$ remains unchanged. We also note that $w(c_1)$ has decreased dramatically, going to .10, since it permits an undesired proof of r. On the other hand, $w(c_2)$ has decreased only slightly, since it contributed to a partial — but not quite complete — proof of r.

At this point, given that $w(c_1) < \sigma$, we are ready to perform a revision at c_1. Our choices are to weaken this clause by adding an additional literal or to delete the clause altogether. Scanning all three exemplars, we see that $N_{c_1} = \{E_1, E_3\}$ and $D_{c_1} = \{E_2\}$. Since the N set is not empty, we chose not to

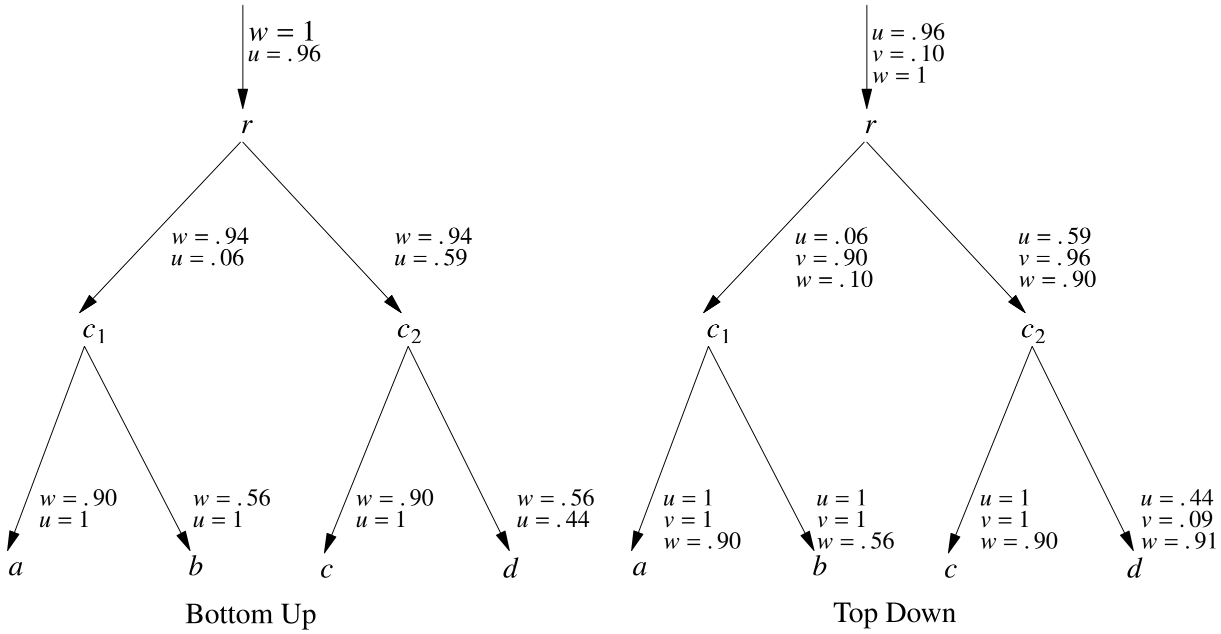

Figure 3: DT graph for sample domain theory after each stage during the processing of exemplar $E_2 = \langle \{a, b, c\}, 0 \rangle$.

delete c_1, preferring instead to weaken the clause via the addition of a new literal q. In order to define this new literal q, we invoke ID3 on $N \bigcup D$, obtaining a DNF expression, e, that separates N from D. Normally, we would define q as the singleton set of clauses $\{c_3\colon q \leftarrow e\}$ and suture this new structure into the DT graph at clause c_1 with $w(q) = \lambda$ and $w(e) = 1$ (see left side of Figure 4). However, in those situations where the size of the DNF expression constructed by ID3 is 1, we can simply promote the literal e directly into clause c_1 with $w(e) = \lambda$ as shown on the right side of Figure 4 (see Footnotes 4 and 5). Recall that in either case we must also reset $w(culprit) = w(c_1) = \lambda$.

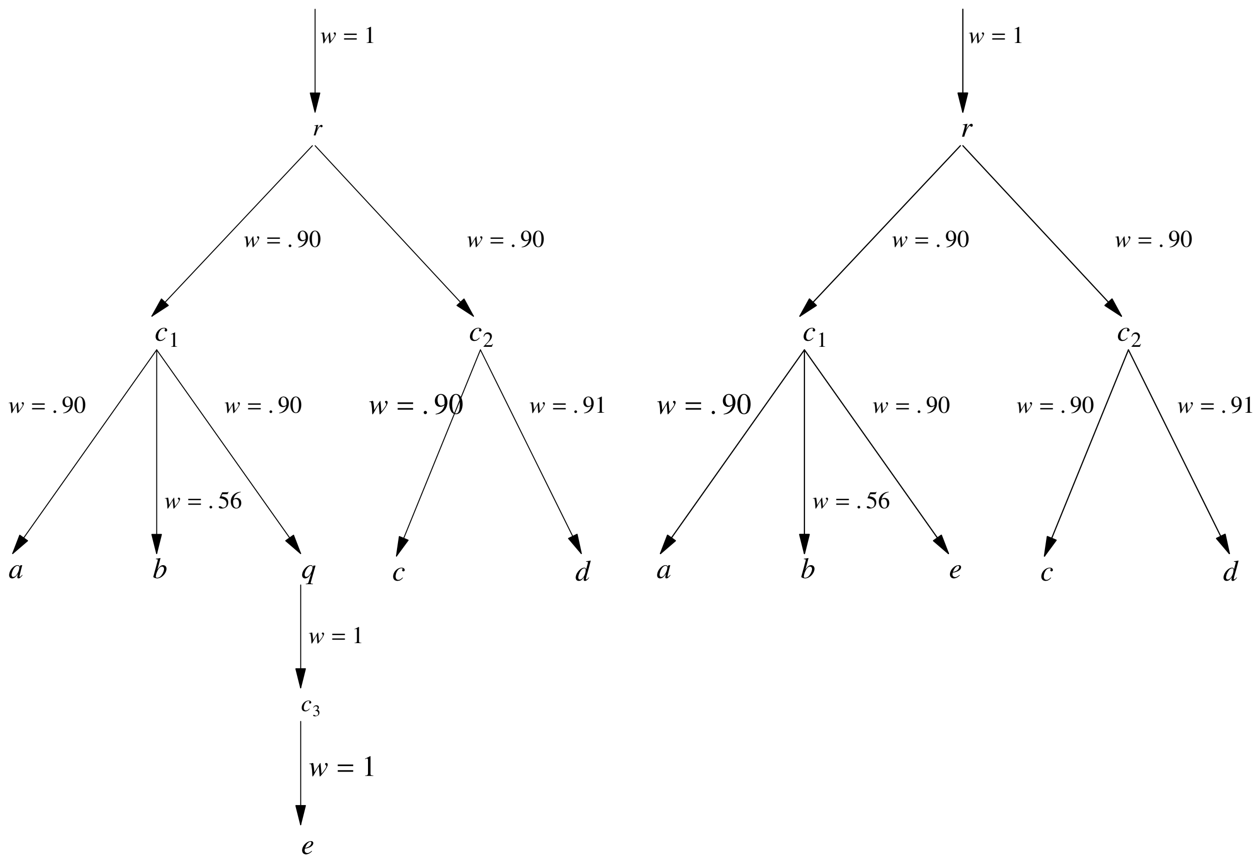

Figure 4: DT graph after addition of new structure below clause c_1 (left) and the simplified version (right).

Once the literal e has been added to c_1, we check for termination only to find that two of the exemplars, E_1 and E_3, are still misclassified. We therefore continue updating weights by processing these two remaining misclassified exemplars in some random order.

Assume we next process the third exemplar, $E_3 = \langle \{a, d, e\}, 1 \rangle$, a misclassified IN exemplar. Figure 5 shows the condition of Δ_Γ during the processing of this exemplar. After processing, both $w(b)$ and $w(c)$ decrease, having each obstructed proofs of r.

At this point, $w(b) < \sigma$, identifying b as the new culprit. Since removing the literal will not compromise any desirable proofs, we can simply delete edge b. PTR0 will now terminate, having classified all examples correctly after the revision. The revised domain theory produced is:

$$c_1 : r \leftarrow a \wedge e$$
$$c_2 : r \leftarrow c \wedge d.$$

If the original revision threshold had been smaller (*e.g.*, $\sigma = 0.10$), $w(b) = 0.19$ would not have been sufficient to cause a revision on this cycle. PTR0 would then have processed the remaining misclassified exemplar E_1. Had E_1 also not been sufficient to force $w(b)$ below σ, PTR0 would have continued to process all misclassified exemplars in some random order after appropriate updating of the σ and λ parameters. Future cycles would have resulted in some additional decrease in $w(b)$, eventually forcing the revision to take place.

2.6. Execution

We now rerun this same example using the Common Lisp implementation of propositional PTR0. There are only a few top-level functions that are directly invoked by the user.

dt-init

The function *dt-init* initializes the DT graph data structure. It should be invoked once before describing the initial domain theory.

defrule

The macro *defrule* is used to establish a definite clause as part of the domain theory. The intent is to mimic in a propositional way the functionality normally attributed to *defrule* in describing the ALPS domain theory.

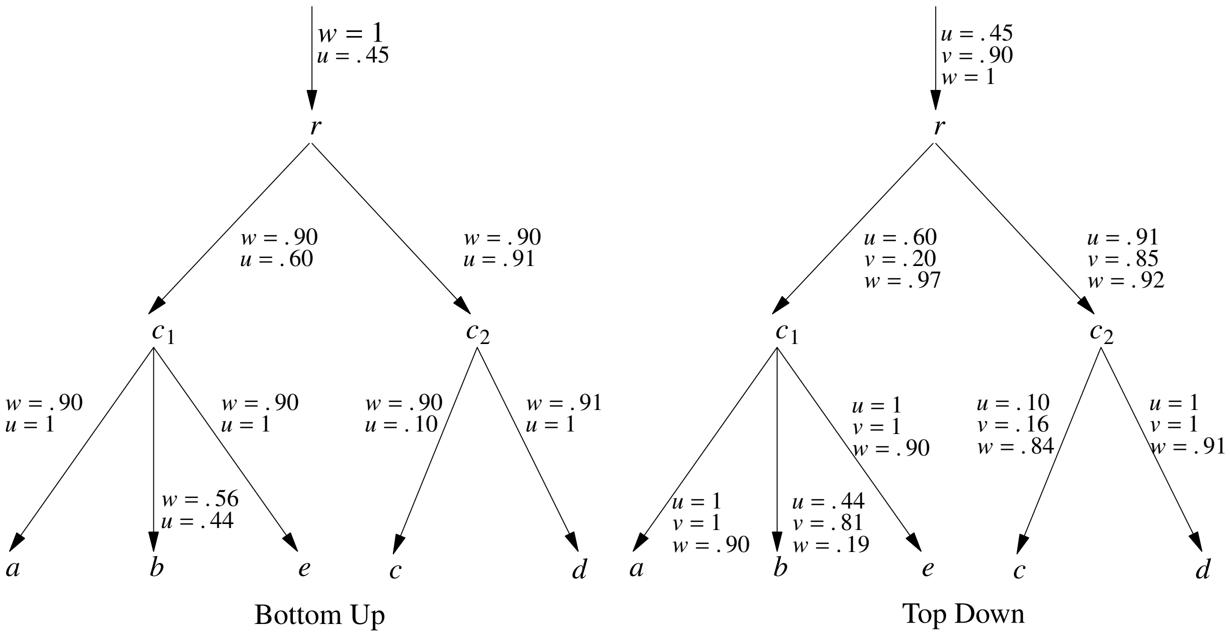

Figure 5: DT graph for sample domain theory after each stage during the processing of exemplar $E_3 = \langle \{a, d, e\}, 1 \rangle$.

show-theory
The function *show-theory* prints out the current domain theory. The optional argument, if t, causes weights to be printed as well.

ptr-trace
The function *ptr-trace*, when called with a non-nil argument, turns on verbose output tracing of PTR's performance.

ptr0 The function *ptr0* runs the propositional PTR0 algorithm on a given set of exemplars. Each exemplar is a tuple of a set of observables and a set of ‘‘in’’ roots, where the set of in roots may of course be empty.[10] Additional optional keyword arguments establish the parameters of the algorithm.

[10] Strictly speaking, to match the more formal discussion of the previous section we would expect the exemplar to consist of tuple $\langle E, \Theta(E) \rangle$, where $\Theta(E)$ is a vector of root truth values. Here, instead of the vector of root truth values, we provide a set of all roots that have 1 as their truth value. This is strictly a matter of convenience.

We can use these commands to mimic the illustration in Figure 5 (note that we have disabled random ordering of exemplars in order to force the same behavior shown above):

```
>(dt-init)                ;; Initialize PTR0
0

>(defrule r <- a b)       ;; Establish rule
NIL

>(defrule r <- c d)       ;; Establish rule
NIL

>(show-theory t)          ;; Print theory with current weights
<1(.90): R <- A(.90) B(.90)>
<2(.90): R <- C(.90) D(.90)>
NIL

> (ptr-trace)             ;; Turn on verbose output
T

> (ptr0 '(((a c e) r)((a b c))((a d e) r))
        :epsilon .1
        :sigma-naught .2
        :lambda-naught .9)      ;; Run PTR0 using appropriate parameters
Cycle 0 <.20:.90> 3 misclassified exemplars
  1: Processing (A C E):(R)
  2: Processing (A B C):NIL
Revising <1>; needed=((A D E) (A C E)), destructive=((A B C))
  Adding dnf ((E)) to clause <1>...special lone clause case
<1: R <- A B E>           ;; Echo revised theory
<2: R <- C D>

  3: Processing (A D E):(R)
Revising <B>; needed=NIL, destructive=((A D E) (A C E))
  Deleting edge <B>
<1: R <- A E>             ;; Echo revised theory
<2: R <- C D>

Complete.                 ;; Done
3                         ;; Total exemplars processed

>(show-theory t)          ;; Print theory with ending weights
<1(.97): R <- A(.90) E(.90)>
<2(.91): R <- C(.82) D(.91)>
NIL
```

2.7. Evaluation

Let us now examine the performance of the PTR0 algorithm on a sample domain drawn from the expert systems literature. The soybean disease diagnosis domain theory was first used by Michalski and Chilausky [Michalski80] in their comparison of machine-induced classification rules with rules provided by a human domain expert. It is most often cited as evidence that machine learning methods can outperform traditional knowledge engineering in the construction of expert systems.

The original domain theory distinguishes between 19 diseases of the soybean plant and comes with a total of 307 exemplars. The theory is expressed in a probabilistic logic, meaning that each literal in the domain theory has associated with it a weight indicating its relative importance. Thus the head of a clause is considered true only if the weighted sum of the antecedents exceeds some built-in threshold. The weights are propagated up the proof, and a diagnosis is made by comparing the weights of the top-level disease predicates. The most probable disease is then returned as the final diagnosis.

A second, reduced, version of the theory that only distinguishes 15 diseases is also reported in [Michalski80]. This reduced theory has previously been used to evaluate theory revision systems: in particular, Ourston uses a slightly modified version of this theory to evaluate the performance of his EITHER system [Ourston91]. In order to permit direct comparison of our results with those for the EITHER system, we use this same theory. Following the procedure they outline, we convert the reduced version of the original probabilistic domain theory into a strictly propositional domain theory by dropping any antecedents with initial weight less than 0.8. The remaining clauses and literals are taken as the initial domain theory, ignoring their associated weights. To maintain consistency with EITHER, each exemplar is forced to correspond to a unique diagnosis by ensuring that vector $\Gamma(E) = \langle \Gamma_1(E) \cdots \Gamma_{15}(E) \rangle$ contains a single 1. We therefore assign $\Gamma_i(E) = 1$ for that i with the largest value for $u(e_{r_i})$, while all other entries in the $\Gamma(E)$ vector are set to 0.

Once converted, the theory consists of 73 rules to distinguish between 15 distinct soybean diseases. Each disease (or DT graph root) relies on a single clause, and the maximum depth of the theory is 3. There are a total of 266 exemplars, where each exemplar consists of an assignment of values to 35 different *symptoms* (*e.g.*, observables) and a single diagnosis. Some of the symptoms are nominal (*i.e.*, ranging over a set of discrete values, for example, $\{green, brown, yellow, \cdots\}$) while others are ordered (*i.e.*, ranging over an ordered set of discrete values, for example, $\{low, medium, high\}$).

A minor extension to the PTR0 algorithm allows us to more appropriately revise range observables (which correspond to leaf literals in the DT graph) in this experiment. Currently, if a leaf literal is selected as the culprit, our only recourse is to delete it since observables are directly present in the input ad thus cannot be revised by the addition of new substructure. This procedure usually leads to overgeneralization (in precisely those situations where the O set is not empty), and the theory must then subsequently be specialized. However, if a range literal is selected as the culprit, it may be possible to avoid overgeneralization and respecialization by extending the range of the literal instead of deleting it altogether. The idea behind range extension as a generalization operator is to alter the endpoints of the range literal until all elements of the I set are covered by the range literal. In the event that members of the O set are also covered, we will again have overgeneralized the theory (but perhaps not as much as would have been the case had the culprit literal been removed), and will subsequently have to specialize it elsewhere.

Initial weights are assigned according to an assignment procedure that takes into account the topology of Δ_Γ [Feldman93a]. The assignment procedure reflects the intuition that elements higher in Δ_Γ should be less susceptible to revision; for example, the root elements that correspond to soybean diseases should probably not be deleted. These default weights are overridden for certain clauses in the original domain theory that are known *a priori* to constitute background knowledge. Such clauses typically test combinations of observables; for example, the predicate *Spring* might be defined as the disjunction of observables *March*, *April*, and *May*. This background knowledge should not be subject to revision by PTR0, and therefore domain theory elements corresponding to background knowledge are assigned an initial weight of 1 [Ginsberg86, Smith85]. However, this technique (*e.g.*, prohibiting refinement of portions of the domain theory altogether) is relatively coarse-grained; PTR0 allows for far more precise control to be exerted on the domain theory [Feldman93b].

We are particularly interested in measuring the *accuracy* of the revised theory as a function of training set size. Accuracy is measured in the traditional manner [Weiss91]. First, a test set of 166 exemplars is randomly selected from the 266 exemplars available. Next, a small training set is randomly selected from the 100 non-test set exemplars and PTR0 is trained on this training set until termination. Finally, the revised theory is tested on all 166 test exemplars. The experiment is repeated for larger training sets, where each successive training set is generated by adding randomly selected training exemplars to the previous training set (*i.e.*, successive training sets are nested). The results reported here are obtained with parameters $\sigma = .10$, $\varepsilon = .01$, and $\lambda = .60$.[11] Figure 6 compares the results of this experiment with similar accuracy results reported by Ourston for the EITHER system [Ourston91]. Since the exact composition of test and training sets for the EITHER experiments is not known, the PTR0 experiment is repeated 20 times, each time with a new random partitioning of training and test sets: the average values over 20 trials are reported in the figure.

As might be expected, test problem set accuracy improves for both PTR0 and EITHER as the size of the training set increases. This is because accuracy depends, in large part, on the domain theory coverage exhibited by the training set. If some portion of the domain theory that requires revision is not exercised by training exemplars but is exercised by test exemplars, then no theory revision system can be expected

---

[11] Additional experiments to gauge the sensitivity of PTR0 to the initial parameter settings are reported in [Feldman93a].

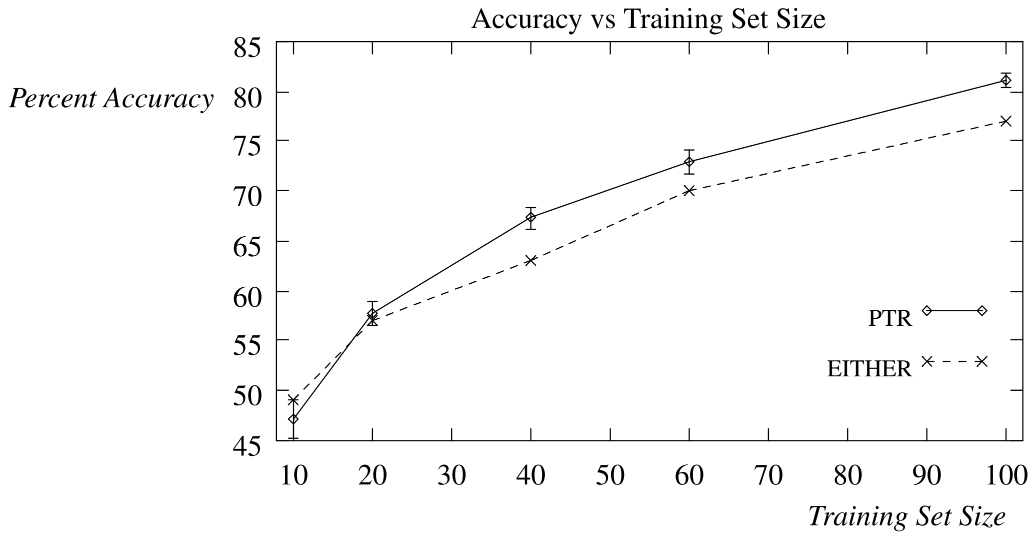

Figure 6: Classification accuracy as a function of training set size for PTR0 and EITHER operating on the soybean domain theory. Each datapoint represents the average value over 20 trials, with error bars representing ±1 standard error.

to revise the unexercised portion of the theory appropriately. As the size of the training set increases, however, so does its coverage, and, concomitantly, classification accuracy on the test set.[12]

Given that EITHER and PTR0 use essentially identical revision strategies, what accounts for the edge in classification accuracy exhibited by PTR0? We believe it is the identification of background knowledge, an example of additional domain knowledge exploited by PTR0. Note that exploiting this additional knowledge need not necessarily entail assigning subjective weights: in this case, it simply

[12] While it is difficult to measure the coverage of the training exemplars, we can get some estimate on the size of the hypothesis space for the soybean domain theory. Recall that the soybean theory has 35 observables. 21 of these observables are boolean, while the remainder have domains ranging in size from 3 to 7. A simple back-of-the-envelope analysis tells us that the hypothesis space contains roughly 164 trillion exemplars: the training sets used here cover only a tiny fraction of this space. For this reason, it is unreasonable to expect perfect accuracy from the revised theory: nevertheless, the accuracy achieved still represents quite a substantial improvement over the original domain theory.

meant identifying certain portions of the domain theory that are not to be considered for revision. The importance of this additional knowledge is clear if one considers that other theory revision systems might consider, for example, dropping the literal *April* from a clause with head *Spring* — contradicting the domain expert's intended semantics of *Spring* — if such a change happens to suit the current configuration of training examples. In contrast, once *Spring* has been identified as background knowledge, PTR0 will never violate the intended semantics of this domain theory element and will be forced to make its revisions elsewhere.

2.8. Remarks

The PTR0 algorithm is an efficient approximation algorithm for the propositional theory revision problem. It exhibits several advantages over previous propositional domain theory revision algorithms.

(1) PTR0 honors the principle of *referential transparency*, retaining the initial structure of the domain theory so that the domain expert can easily track changes made by PTR0.

(2) PTR0 exploits additional knowledge provided by the domain expert in the form of weights to speed the revision process.

(3) PTR0 is an efficient algorithm that is well-suited for implementation on a fine-grained SIMD machine.

(4) Unlike neural network based techniques, formal proofs of termination and convergence exist for the PTR0 algorithm: proofs which rely on the theoretical underpinnings of the PTR0 algorithm.

(5) Natural extensions to the PTR0 algorithm allow for handling noise in the classification of exemplars.

We have performed extensive experiments with the PTR0 system in a variety of domains; the interested reader is referred to [Feldman93a] and [Koppel94].

3. PTR1: Probabilistic Theory Revision for Relational Theories

While the PTR0 algorithm defines the state of the art for propositional theory revision systems, most interesting theories are in fact not propositional, and are more naturally given in terms of a more expressive first-order logic. For example, the transportation scheduling domain theory used by the ALPS system is expressed in a first-order definite-clause framework, replete with negation-as-failure and recursion, yet less expressive than full first-order logic due to its definite-clause restriction. Indeed, it is precisely this sort of definite-clause logic that forms the representational basis of Prolog, which has in the past been used to construct expert systems with some success. For this reason, it seems reasonable to extend the PTR framework to just this sort of logical framework.

Here we describe PTR1, a natural extension of the PTR0 algorithm suitable for relational theories, an important subset of first-order logic. Previously, we had used a reconstruction of FOIL [Quinlan90] in the place of the propositional inductive learning system ID3 used in PTR0.[13] However, we have since decided to replace FOIL with an inductive logic algorithm that takes advantage of the DT graph structure.

3.1. Beyond Propositional Calculus

There are several intrinsic differences between the propositional system and a first-order system. The first difference is representational; the head and body of a clause are no longer composed solely of propositions like p, but rather are allowed to be atomic formulae replete with variables, such as $p(X,Y)$. Thus while PTR0 has rules that look like $r \leftarrow pq$, PTR1 will have rules that look like $r(X,Y) \leftarrow q(X)q(Y)$ (where tokens beginning with upper case letters represent universally quantified logic variables according to the commonly used Prolog convention).

A second major difference is in the semantics of negation-as-failure. In the propositional case, failing to find a proof for a proposition was sufficient reason to assume that the proposition was false. The semantics are somewhat different in the relational case. For example, consider the atomic formula $p(X)$. An appropriate interpretation has X as an existentially quantified logic variable, meaning that *any* binding of X is acceptable for the proof of $p(X)$. However, the logic variable in the atomic formula $\neg p(X)$ is universally quantified, meaning that to prove $\neg p(X)$, $p(X)$ must not be provable for *all* possible values of

[13] FOIL is a relational version of ID3, the propositional inductive learning system used in PTR0. FOIL takes extensional definitions of exemplars and produces a relational DNF expression of the concept. Note that we could just as easily have used some other first-order component such as CHAM [Kijsirikul91] or FOCL [Brunk91].

X.

A third major difference is that each domain theory now includes some fixed set of axioms, such as $q(a)$ or even $q(Z)$. These axioms replace the use of observable sets included with each exemplar. Accordingly, each exemplar now consists of only a query term, such as $r(a)$ or $r(W)$, and the expected outcome for that query (note an additional difference with respect to the propositional case: the expected outcome is given only for the query in question, and not for a vector of root propositions as in the propositional case).

Finally, another difference is that rules of this form might now form recursive constructs, where an atomic formula in the body of a clause might "match" the atomic formula that constitutes the head of that same clause. Such recursive constructs are meaningless in the propositional case, but have meaning here given the addition of logic variables which can take on different bindings. For example, the rule $r \leftarrow p, r$ is clearly meaningless, while the rule $r(X, Y) \leftarrow q(X, Z), r(Y, Z)$ can make perfect sense. Note also that, contrary to this simple example, the recursion might span several clauses and be difficult to detect.

To summarize, the four main problems encountered in extending PTR0 to first-order logic are:

the presence of variables in the rules,
the use of negation-as-failure with variables,
the changes in the structure of exemplars, and
the use of recursive rules.

The PTR1 algorithm handles the first three problems cleanly and operates properly for recursion-free definite-clause theories.[14]

[14] We believe that the framework can bear further improvement and extension to handle recursion as well; this is a topic of continuing interest and research.

3.2. Relational DT Graphs

In this section, we briefly describe the changes made to the underlying DT graph representation to support PTR1. These extensions are relatively straightforward; as before, the clause c_i: $H_i \leftarrow B_i$ represents the assertion that the atomic formula H_i is implied by the conjunction of atomic formulae in B_i. Since we are now dealing with atomic formulae instead of propositions, however, we must label each literal or clause edge with a atomic formula which may contain logic variables. The process is quite straightforward. As clauses in the original domain theory are parsed to construct the DT graph, each clause is *standardized apart* to ensure there are no variable naming conflicts. Thus variable names are shared between a clause edge and its immediate child literal edges, but not among, for example, clause edges or unrelated literal edges.

An example should help make this clear. Consider the simple domain theory:

c_1: $m(X, Y) \leftarrow p(X)$
c_2: $r(X) \leftarrow s(X)$
c_3: $s(X) \leftarrow t(X)$
c_4: $u(X, X) \leftarrow p(X) \wedge \neg(q(X))$
c_5: $p(a)$
c_6: $p(b)$
c_7: $q(a)$
c_8: $q(c)$
c_9: $t(d)$

where we adhere again to the Prolog convention of using capital letters for logic variables. Clauses c_5 through c_9 correspond to axioms of the domain theory.

Figure 7 shows the DT graph for this simple domain theory. Like the propositional case, each unique predicate gets a root pseudo edge directed towards the node of the DT graph representing that particular predicate. As in the propositional case, each root pseudo edge gets unit weight, since it represents the entry point to the theory for a given atomic formula and should therefore remain invariant.

Each clause in the domain theory corresponds to a single clause edge and some number of literal edges connecting the unique clause node to the predicate nodes corresponding to the predicates of the wffs labeling those edges. Some clauses have no literal edges; these clauses correspond to axioms of the theory (see, for example, c_5 in Figure 7). Recursion is represented in the obvious way, although the current version of PTR1 is restricted to recursion-free relational theories.

3.3. Exemplars

Recall that in PTR0, an exemplar consists of a pair $\langle E, \Theta(E)\rangle$, where E is a set of observable facts (corresponding to leaves of the DT graph) and $\Theta(E)$ is a vector of expected outcomes for each root of the DT graph. For PTR1, a few changes are in order.

In a relational domain, we are typically providing a number of atomic formulae (possibly containing universally quantified logic variables) as queries, and asking whether or not any instances of these formulae have legal proofs given with the domain theory. Here, the domain theory typically contains a number of axioms that correspond to observables for the domain. These observables are seen as invariant, that is, they are fixed and do not change from one exemplar to the next; to reflect this status, then, we initially assign unit weight to clause edges in the DT graph corresponding to axioms (just as we do for root edges).

In another clear departure from the propositional case, note that we are no longer looking for proofs of propositions in some preestablished root set; a query might consist of *any* atomic formula, not just one of a preidentified set. Thus an exemplar can tell us nothing about the truth status of any other atomic formula other than the original query.

To reflect these differences, then, we define an exemplar, $\langle \omega, \Theta(\omega)\rangle$, to be a tuple consisting of a universally-quantified atomic formula ω and a single target outcome $\Theta(\omega)$. For the simple theory of Figure 7 we might have the following exemplars:

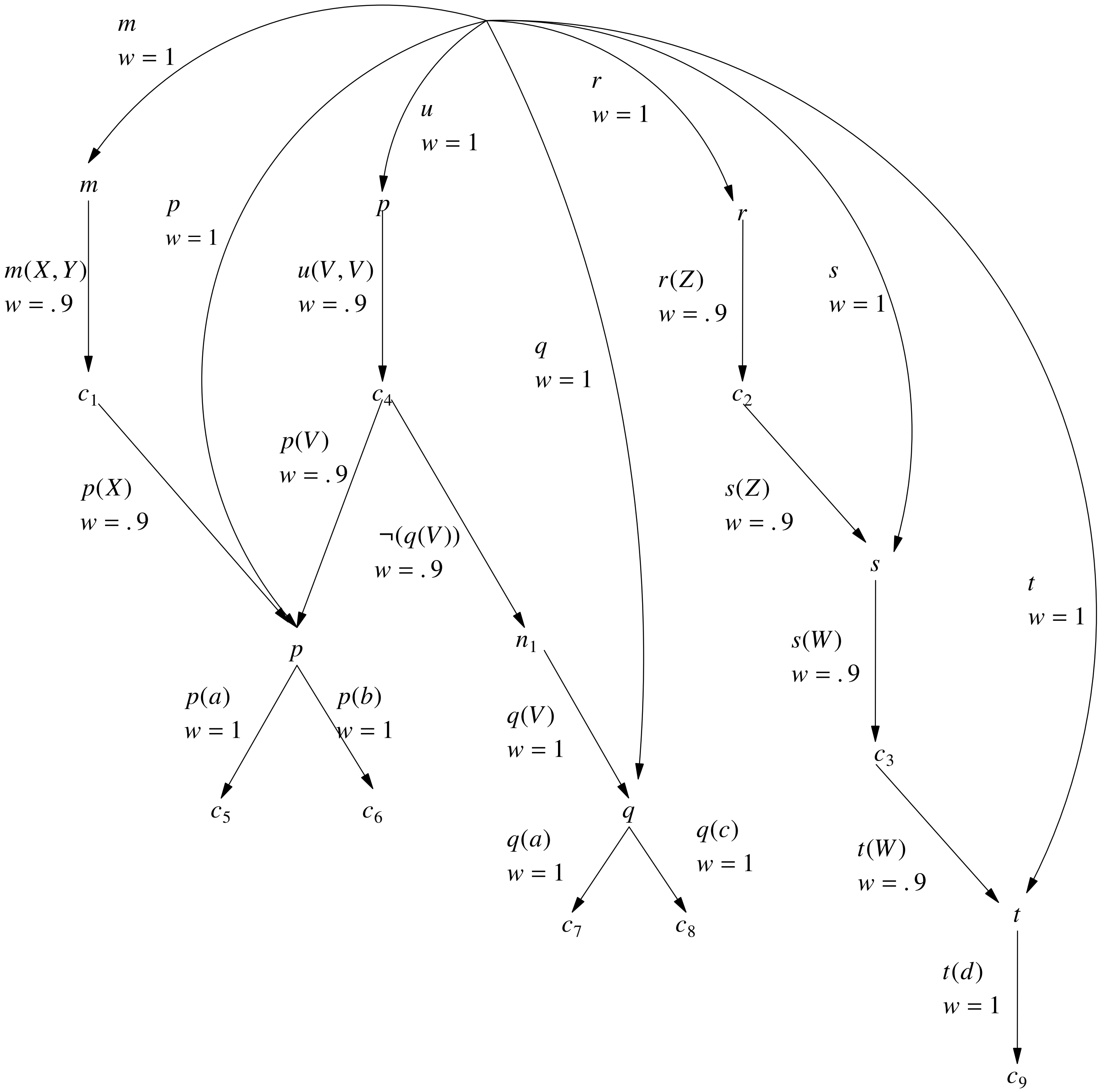

Figure 7: DT graph for sample first-order domain theory.

$\langle m(a,a),1\rangle$
$\langle m(a,b),0\rangle$
$\langle r(d),1\rangle$
$\langle r(e),0\rangle$
$\langle s(d),0\rangle$
$\langle s(e),1\rangle$
$\langle u(b,b),1\rangle$
$\langle u(a,e),1\rangle$
$\langle u(a,d),0\rangle$

In addition to the differences in exemplar structure just described, it is worth noting that determining which exemplars are misclassified is much more difficult than in the propositional case. Recall that in the propositional case, a simple algorithm based on the bottom-up weight updating procedure could be used to determine the truth value of any root given a set of observed propositions. In the first-order case, however, it is necessary to use a first-order inference engine, much like Prolog, to determine the truth value of a universally-quantified query formula. For the exemplars just given, $\langle m(a,b),0\rangle$, $\langle s(d),0\rangle$, $\langle s(e),1\rangle$, and $\langle u(a,d),0\rangle$ are misclassified since none of these query forms are provable using the domain theory described previously.

3.4. Update Algorithms

The updating method at the core of PTR0 assumes that provided exemplars unambiguously assign truth values to each observable proposition. In first-order theory revision, the truth of an observable predicate typically depends on variable assignments that may be implicit in the query. Thus, in order to apply this approach to the revision of relational theories it is necessary to determine some ‘‘optimal’’ variable assignments on the basis of which probabilities can be updated.

Our approach is to make some structural changes to the DT graph, allowing each edge to retain multiple copies of the intermediate $u(e)$ and $v(e)$ values computed in the course of the weight updating algorithm. These multiple copies are indexed on the basis of some binding assignment for the logical variables at that edge.

We proceed as follows. We alter the bottom-up updating procedure to start by ‘‘forcing’’ a copy of the query atomic formula down the DT graph, beginning at the corresponding root edge. Intuitively, the purpose of this ‘‘forcing’’ operation is to censor those parts of the DT graph which are not relevant to the current exemplar because of binding constraints implicit in the exemplar's query itself. As each clause edge is encountered, PTR attempts to unify the query with the current clause, and, if successful, the resulting binding list is passed as an argument (instead of the atomic formula) to the recursive call of the bottom-up procedure on the child literal edges of the current edge (if the unification is not successful, the current edge is considered ‘‘inactive’’ for this particular bottom up process and is marked accordingly). As a literal edge is encountered, the binding list received from the parent edge is applied to the label of the edge, and the resulting formula is then passed (instead of the binding list) to the recursive call of the bottom-up procedure on the child clause edges of the current edge. Negation edges are handled just as literal edges; the binding list is applied to the edge label, but it is also passed on unchanged to the single child literal edge.

The leaves of the DT graph are also clause edges; once we reach the leaves, we treat them just like any other clause edge. If the unification performed there is successful, then the appropriate u value is computed, cached at the edge indexed by atomic formula, and returned to the parent edge. At this point we begin the normal bottom-up propagation, computing u values as usual at literal edges, caching them at the edge (indexed by binding list), and returning these u values to the parent edge. The u value obtained at the original root edge corresponds to the proof flow for the given query with appropriate binding constraints enforced. For example, for the misclassified query $\langle m(a,b), 0\rangle$, a proof flow of $u = 0.90$ is computed at the root edge for predicate m. For the misclassified query $\langle s(e), 1\rangle$, a proof flow of $u = 0.09$ is computed at the root edge for predicate s.

The top-down process is modified in a similar way. The atomic formula from the query is used to select the appropriate root edge as a starting point for the top-down process. We then proceed recursively, computing a value of v for every value of u cached at the edges encountered. As in the propositional case, each v value relies on the notion of a most affected parent edge (Equation 4); however, the definition of the most affected parent edge must now, of course, be sensitive to binding constraints. This means that a different parent edge might be used for the computation of each v value at the same edge, depending on the index used for the corresponding u value (to identify the most affected parent edge, we only consider parent u and v values whose indices are consistent, in the unification sense, with the child u index).

Once we have computed v values for every u value cached at the edge, we proceed with the computation of the edge w value. Unlike computation of u and v, only a single value of w is computed for each edge. Recall Equation 5 computes w as a function of the previous w and the ratio of the v and u values. Since we do not want to maintain multiple values of w, we will approximate the correct, binding-sensitive, behavior by using a maximally conservative strategy. Our approximation involves using the smallest available ratio of v and u for the computation of w at each edge. In this way, each edge's w value changes as slowly as possible. Of course, the use of such an approximation strategy poses some problems; perhaps most obviously the fact that termination and convergence can no longer be guaranteed. But empirical evidence suggests that this is not a problem from a practical perspective; in our experience, the algorithm has always converged.

3.5. Inductive Algorithm

As we have just seen, PTR1 operates in the same fashion as PTR0, locating a culprit edge that needs to be revised. Once a culprit edge has been located, N and D sets are computed just as in the propositional case. As stated earlier, a reconstruction of the FOIL algorithm (essentially a relational version of ID3) had previously been used at this point to find a suitable DNF expression to suture into the DT graph at the culprit. However, using the FOIL algorithm presented several problems. FOIL requires the extensional definition of the input exemplars, meaning that we lose some of the advantages that arises in the culprit identification stage of PTR1. For example, to properly operate with FOIL, we must produce all possible instantiations of the exemplars in the N and D sets. Far better would be to exploit the structure of the DT graph in some way to find the appropriate new structure to correctly qualify a culprit edge. So FOIL has now been replaced with an inductive learning algorithm that takes advantage of the DT graph structure.

Unlike ID3 and FOIL which try to find a DNF expression that completely separates the N and D sets, our algorithm greedily searches for the single literal that ‘‘best’’ separates the two sets. This greatly simplifies the algorithm since it does not have to completely separate the two sets. Should multiple literals need to be added, we depend on PTR1 to once again find the current culprit (or possibly the newly added literal) to be a culprit in the future so that additional literals may be added. Thus, a completely new DNF expression may be built over several revisions, through the guidance of PTR1’s ability to find a culprit. Additionally, it should be noted that backtracking is inherently implemented since PTR1 may find any of the new literals to be culprits and may delete them if no exemplars are in need of the literals any longer.

Throughout this discussion, the example that was started above and whose execution is shown in one of the following sections will be repeatedly used. Hopefully, by providing a concrete example, some of these concepts will be made clearer.

To find the single literal that ‘‘best’’ separates the N and D sets, we begin by determining what argument bindings each of the relevant exemplars produce at the culprit’s clause. Unlike in the propositional case which did not have variables, argument bindings are a major distinguishing feature between exemplars, as there may be many IN and OUT exemplars that have the same root but different argument bindings. Whether the culprit is a clause head or a member of a clause body, all of the arguments of the culprit’s clause head are available to use as arguments to any newly added literal. To determine the argument bindings at the culprit’s clause head, we take advantage of the bottom-up algorithm’s caching of argument bindings as indexing to the u values. Thus for each relevant exemplar, we perform a bottom-up and store the argument bindings found at the culprit's clause head.

In the example, after PTR1 process the first misclassified exemplar, it finds the clause $m(X, Y)$ to be the culprit, $N = \{m(a, a)\}$, and $D = \{m(a, b)\}$. The culprit arguments associated with the N set clearly are (a, a) and those associated with the D set are (a, b). It should be noted that, unlike this example, the arguments at the culprit's clause head are not always the arguments to the exemplar.

It is these argument bindings that in some sense now become our needed and destructive exemplars except they are missing a root. A newly added literal will not necessarily see the argument bindings of the N and D exemplars, but rather the argument bindings that the bottom-up algorithm finds at the culprit's clause head. When we eventually add the new literal, its arguments will all be variables (many corresponding to the variables of culprit's clause head). While a new literal could be added with some of its arguments already instantiated, this was not done since it would greatly increase the search complexity.

Now with the culprit's argument bindings in hand, we wish to find a literal that will be true for all of the argument bindings that were generated from the needed exemplars and false for all of the argument bindings from the destructive exemplars or vice versa since we can use the negation of the literal. However, the literals that are available to us may have different numbers of arguments, and we may not want to pass the arguments to a new literal in the same order as we find them at the culprit's clause head. In fact, we may wish to duplicate arguments or even create new variables. For example, with $m(X, Y)$ as the culprit, we may wish to add such literals as $u(Y, Y)$, $q(X)$, $q(Y)$, or $u(X, Z)$ to the clause. So now we are not only looking for the root that best separates the N and D sets, but also the ordering of arguments (some of which may even be new variables). It should be noted that a literal whose arguments are all new variables could never separate two sets of exemplars since the literal must always be true or always false; thus, only argument orderings with at least one variable from the culprit's clause head are considered.

Next, we need some measure of how well a root and an argument ordering separates the N and D sets. The proof flows computed by the bottom-up algorithm give a very good approximation of the probability that a particular exemplar is true. Thus, we choose to look for the root and argument ordering that "best separates" the proof flows of the needed and destructive culprit argument bindings. So, for each root and argument ordering, we perform a bottom-up with each of the needed and destructive culprit argument bindings. Keeping the data for the needed and destructive sets separate, we determine the average, maximum, and minimum proof flow over the bindings for each of the root and argument orderings.

At this point we must define exactly what we mean by "best separates the proof flows." First we look for any literals that completely separate the needed and destructive sets; that is, the minimum proof flow of one set is greater than the maximum proof flow of the other set. If such literals exists, we choose the one that whose proof flow averages have the greatest difference between the needed and destructive sets. If a complete separation does not exists, we simply look for the literal whose proof flow averages have the greatest difference.

Returning to the example, recall that for the culprit $m(X, Y)$, (a, a) are the culprit bindings associated with the N set and (a, b) are the culprit bindings associated with the N set. For simplicity, let us work out the separation that the root q provides. First, there are only two possible orderings of the argument bindings for q since it has only one argument, namely $q(X)$ and $q(Y)$. So, for the N set, we perform bottom-up's on $q(a)$ and $q(a)$, each returning a proof flow of 1 (since $q(a)$ is an axiom). Likewise, for the D set, we perform bottom-up's on $q(a)$ and $q(b)$, returning proof flows of 1 and 0. Therefore, $q(X)$ has no separation since both the N set and the D set have proof flows of 1; however, $q(Y)$ has a perfect separation since the N set has a proof flow of 1 and the D set has a proof flow of 0. In fact, it is $q(Y)$ that the algorithm finds to add to the culprit's clause. It should be noted that this is a very simple example; many times there are multiple culprit bindings associated with each set (and their number may not be equal) as well as many more possible argument orderings for the root.

Finally, this new literal is added to the DT graph as discussed in the following section. Note that the sign of the difference of averages indicates the sense of the new literal -- whether the new literal should simply be added or whether its negation should be added.

To ensure that the same literal does not repeatedly get added to the same culprit, all literals that appear in the culprit's clause are prevented from being added. Note that this is very specific: only literals of the same sense (whether they are negated or positive) and of the exact same arguments are prevented from being added. Clearly a rule such as $m(X,Y) \leftarrow p(X)p(X)$ is not logically any different from $m(X,Y) \leftarrow p(X)$. So this only prevents adding redundant literals and really bad recursion (where the head appears in the body with the same arguments).

3.6. Revision Operators

Besides the four revision operators discussed in the propositional case, we have added two additional revision operators for use in the relational case. The six relational revision operators are:

add a literal to an existing clause,
delete an existing clause,
add a new clause,
delete a literal from an existing clause,
replace a literal with its definition, and
add a new clause stub.

As was noted in the propositional case, if the N set is empty, the culprit (whether a clause or a literal) may be deleted. If the N and D sets are non-empty, then except in the special case mentioned below the inductive algorithm described above is run. If the culprit is a clause then the new literal is simply sutured into the clause just as was discussed for the special case of a single literal in propositional case. If the culprit is a literal, a new symbol is generated with the same arguments as the culprit's clause head, and this new symbol replaces culprit. Two new bodies are then added to the new symbol -- namely, the culprit and the new literal from the inductive algorithm. This accounts for the first four revision operators in a very similar manner to how they were used in the propositional case.

3.6.1. Replacing a Literal with its Definition

During the discussion of the inductive algorithm, it was noted that the bindings of the culprit's clause head arguments are determined for each of the relevant exemplars and are used to try to separate the N and D sets. However, it is quite possible for two different exemplars to produce the same arguments at the culprit's clause head. If one of these exemplars is a member of the N set and the other a member of the D set, it is clear that no literal could separate the N and D sets since a literal can not produce different results from the same arguments.

Once again continuing with our example, the next culprit PTR1 finds is the clause $s(X)$ with $N = \{s(e), r(d)\}$ and $D = \{s(d)\}$. The relevant clauses are:

$$c_2 \colon r(X) \leftarrow s(X)$$
$$c_3 \colon s(X) \leftarrow t(X)$$

Then, the culprit argument bindings associated with the N set are $\{(e), (d)\}$, and those associated with the D set are (d). Clearly the argument (d) is duplicated across both sets. To avoid this problem, we would like to replace $s(X)$ in c_2 with its definition, $t(X)$, so that both clauses could be modified independently. This modification would make c_2 become $r(X) \leftarrow t(X)$.

For a given set of culprit argument bindings, let *NS* be the set of all needed exemplars that produce such culprit argument bindings ($r(d)$ in our example), and likewise let *DS* be the set of all destructive exemplars that produce such culprit argument bindings ($s(d)$ in our example). If all of the roots of the *NS* exemplars do not appear as any of the roots of the *DS* exemplars, then we wish to separate these roots from each other so that each may be modified independently. This separation can be done by replacing the appropriate literals with their definitions as was shown in the above example.

Next we must determine which literal or literals should be replaced with their definitions. We begin by noting that only literals farther up in the DT graph from the culprit need to be considered since they are the only ones through which the culprit receives its argument bindings. The literals that should be replaced with their definitions are those through which only the needed or only the destructive arguments pass through, but whose definitions have needed and destructive arguments passing through. In the example, only the needed arguments pass through $s(X)$ in c_2, but both the needed and destructive arguments pass through $t(X)$ which is $s(X)$'s definition. As we used the bottom-up algorithm to determine the culprit's argument bindings, we can use the bottom-up algorithm to trace the flow of arguments through the DT graph from the exemplar's root to the culprit. The bottom-up algorithm's caching of u-values and bindings serve as the trace.

Thus, we proceed to perform a bottom-up on all of the needed exemplars and then cache all u-values found from the culprit upwards into a temporary slot. Next we perform a bottom-up on all of the destructive exemplars. Then we begin recursively walking up the DT graph from the culprit following the flow of both the needed and destructive arguments. When an edge is encountered through which only needed or only destructive arguments pass, it is replaced with its definition.

However, if a root in the *NS* set also appears as a root in the *DS* set, then it is impossible to separate these exemplars from each other by simply replacing a literal with its definition. We first try to ignore these exemplars by deleting them from the needed and destructive sets and proceeding as normal. That is, if there remain exemplars that produce the same culprit arguments, we replace literals with their definitions; otherwise, if the new destructive set is non-empty, then the inductive algorithm is run on the new needed and destructive sets. If proceeding as normal with these new N and D sets is impossible, but there are additional exemplars (with different root(s)) that produce the same culprit arguments as the ones with the same root, then we will try to separate the exemplars with the same root from those with different roots. This is done just as we separated the needed and destructive exemplars producing the same culprit arguments by replacing certain literals with their definitions. Should all of these fail, we have no other choice except to delete the culprit and let PTR1 correctly add the appropriate literals.

3.6.2. Adding a New Clause Stub

With PTR1, it is possible for a misclassified exemplar to not be able to make any edge culpable. This can be caused by several distinct reasons. The domain theory may not contain any rules (although it might have axioms) for a specific root of a misclassified exemplar. Or, the only rules for a specific root might all be unit weighted so that they are unmodifiable. Also, a misclassified exemplar might not be able to unify with any of the existing rules for its root. In our example, the misclassified exemplar $u(a, e)$ can not make anything culpable because the only rule for u is one whose head requires both arguments to be the same: $u(X, X) \leftarrow p(X) \wedge \neg(q(X))$.

More specifically, the detection algorithm proceeds by forcing a copy of the query atomic formula down the DT graph as was done in the bottom-up algorithm. If any edge that the query will unify with is not unit weighted, then the domain theory is culpable with respect to the exemplar. Otherwise, if all edges that will unify with the query are unit weighted, then the domain theory is inculpable with respect to the exemplar.

Once it has been detected that the domain theory is inculpable with respect to an exemplar, PTR0 must make it culpable in some way. This is done by adding a new clause stub which basically is a new rule without any antecedents and whose consequence is simply the exemplar with all of its arguments as new distinct variables. Thus in the example, the rule $u(X, Y) \leftarrow nil$ would be added. This new rule stub is added with lambda weight unless lambda equals one in which case it is added with a default weight of 0.9. In some sense this is like adding a ‘‘non-certain axiom,’’ but we fully expect PTR1 to later find this clause to be a culprit and to add the appropriate literals. This basically amounts to predicate invention, allowing PTR1 to add literals one at a time.

3.7. Execution Revisited

An example should help make this clear. We will use the prototype Common Lisp implementation of relational PTR1 to process the example described above. Our current implementation is integrated with the ALPS Lisp Inference Engine, but since PTR1 does not yet support recursion, we are limited to recursion-free definite-clause theories with negation. Since the ALPS Lisp Inference Engine is case insensitive, we will denote logic variables with a leading "?" rather than relying on uppercase.

The implementation of PTR1 retains many of the same functions as the propositional implementation, with just a small number of modifications.

dt-init
: The function *dt-init* is unchanged.

defrule
: The macro *defrule* now supports recursion-free definite-clauses with negation. The format is identical to that used normally in defining domain theories for the ALPS inference engine.

show-theory
: The function *show-theory* is unchanged.

ptr-trace
: The function *ptr-trace* is unchanged.

ptr1
: The function *ptr1* runs the PTR1 algorithm on a given set of exemplars. Each exemplar is a tuple consisting of an atomic formula and an expected outcome. Additional optional keyword arguments establish the parameters of the algorithm, and are identical to those for *ptr0*.

We can now examine the operation of the PTR1 algorithm.

```
>(dt-init)                            ;; Initialize PTR1
0

>(defrule (m ?x ?y) <- (p ?x))        ;; Establish first rule.
<(M ?1 ?2)>

>(defrule (r ?x) <- (s ?x))           ;; Establish second rule.
<(R ?3)>

>(defrule (s ?x) <- (t ?x))           ;; Establish third rule.
<(S ?4)>

>(defrule (u ?x ?x) <- (p ?x) (not (q ?x)))   ;; Establish fourth rule.
<(U ?5 ?5)>

>(defaxiom (p a))                     ;; Establish axiom.
<(P A)>
```

```
>(defaxiom (p b))                        ;; Establish axiom.
<(P B)>

>(defaxiom (q a))                        ;; Establish axiom.
<(Q A)>

>(defaxiom (q c))                        ;; Establish axiom.
<(Q C)>

>(defaxiom (t d))                        ;; Establish axiom.
<(T D)>

>(ptr-trace)                             ;; Turn on verbose output.
T

>(ptr1 '(((m a a) . :IN) ((m a b) . :OUT)
         ((r d) . :IN)   ((r e) . :OUT)
         ((s d) . :OUT)  ((s e) . :IN)
         ((u b b) . :IN) ((u a e) . :IN) ((u a d) . :OUT)))
Cycle 0 <.10:.90> 4/9 exemplars
  1: Processing (M A B) :IN, should be :OUT

*** Revising <(M ?1 ?2)>
   with 1 NEEDED=((M A A))
   and  1 DESTRUCTIVE=((M A B))
*** Adding expression (Q ?2) to clause <(M ?1 ?2)>
*** 3 misclassified exemplar(s) remain.

  2: Processing (S D) :IN, should be :OUT

*** Revising <(S ?4)>
   with 2 NEEDED=((S E) (R D))
   and  1 DESTRUCTIVE=((S D))
*** Replacing <(S ?3)> with its definition (special 1 literal case).
*** 3 misclassified exemplar(s) remain.

  3: Processing (S E) :OUT, should be :IN

*** Revising <(T ?4)>
   with 0 NEEDED=NIL
   and  1 DESTRUCTIVE=((S E))
*** <(T ?4)> dropped.
*** 2 misclassified exemplar(s) remain.

  4: Processing (U A E) :OUT, should be :IN
*** Making (U A E) culpable.
*** 2 misclassified exemplar(s) remain.

Cycle 1 <.10:.90> 2/9 exemplars
  5: Processing (S D) :IN, should be :OUT
```

```
*** Revising <(S ?4)>
    with 1 NEEDED=((S E))
    and  1 DESTRUCTIVE=((S D))
*** Adding expression (NOT (T ?4)) to clause <(S ?4)>
*** 1 misclassified exemplar(s) remain.

   6: Processing (U A D) :IN, should be :OUT

*** Revising <(U ?107 ?108)>
    with 1 NEEDED=((U A E))
    and  1 DESTRUCTIVE=((U A D))
*** Adding expression (NOT (T ?108)) to clause <(U ?107 ?108)>
*** 0 misclassified exemplar(s) remain.

Success.
6
>(show-theory)                          ;; Print revised theory.
<2[.90]: (M ?1 ?2) <- (P ?1)[.90] (Q ?2)[.90]>
<5[.90]: (R ?3) <- (T ?3)[.90]>
<7[.90]: (S ?4) <- (NOT (T ?4))[.90]>
<10[.90]: (U ?5 ?5) <- (P ?5)[.90] (NOT (Q ?5))[.90]>
<17[.90]: (U ?107 ?108) <- (NOT (T ?108))[.90]>
```

4. Concluding Remarks

Our preliminary work with the PTR1 algorithm is quite encouraging; we believe that it is possible to attain the same level of revisional ability with PTR1 on relational theories as exhibited by PTR0 for propositional theories. We also believe that PTR1 can eventually be extended to definite-clause theories with recursion. However, there are still a number of unresolved issues, both practical and theoretical, which must be addressed before this will be possible.

The final major problem with PTR1's ability to handle relation theories is that bottom-up and top-down do not correctly handle clauses with variables in the body that do not appear in the head. For example, consider a rule such as $grandparent(X, Y) \leftarrow parent(X, Z) parent(Z, Y)$. Bottom-up and top-down heuristically only require there to be some Z in each of the parent literals but the Z does not have to match. Thus, bottom-up would give $grandparent(john, sue)$ a high proof flow even if $parent(john, mary)$ and $parent(sam, sue)$ were the only axioms in the domain theory.

Second, top-down and bottom-up algorithms must be modified so as to operate correctly with recursive rules. The problems here are obvious; there must be some notion of termination that does not require enumerating all possible paths through a recursive theory, which may, of course, be infinite.

Third, while proofs of termination and convergence are likely to be much more difficult to construct than in the propositional case, it would be well worth the time spent to see if the original proofs can be extended to cover the relational case using the approximation strategy just described.

Fourth, some additional effort should be expended on extensions to the relational scheme, allowing, for example, axioms that might change with exemplars, or axioms that are not considered fixed (*i.e.*, are allowed to have non-unit weights initially) to allow for noise (this latter extension should entail no changes to the algorithm as it stands).

Finally, perhaps most important of all, extensive empirical testing should follow, including comparison to existing work on first-order theory revision as well as work in the general area of inductive logic programming.

References

[Brunk91]
C.A. Brunk and M.J. Pazzani, "An Investigation of Noise-Tolerant Relational Concept Learning Algorithms," *Proceedings of the Eighth International Machine Learning Workshop* (June 1991), pp. 389-393.

[Feldman93a]
R. Feldman, *Probabilistic Revision of Logical Domain Theories*, Ph.D. Thesis, Department of Computer Science, Cornell University, Ithaca, NY (1993).

[Feldman93b]
R. Feldman, M. Koppel, and A.M. Segre, "The Relevance of Bias in the Revision of Approximate Domain Theories," *Working Notes of the 1993 IJCAI Workshop on Machine Learning and Knowledge Acquisition: Common Issues, Contrasting Methods, and Integrated Approaches* (August 1993), pp. 44-60.

[Ginsberg86]
A. Ginsberg, "A Metalinguistic Approach to the Construction of Knowledge Base Refinement Systems," *Proceedings of the Fifth National Conference on Artificial Intelligence* (August 1986), pp. 436-441.

[Kijsirikul91]
B. Kijsirikul, M. Numao, and M. Shimura, "Efficient Learning of Logic Programs with Non-determinant, Non-discriminating Literals," *Proceedings of the Eighth International Machine Learning Workshop* (June 1991), pp. 417-421.

[Koppel94]
M. Koppel, R. Feldman, and A.M. Segre, "Bias-Driven Revision of Logical Domain Theories," *Journal of Artificial Intelligence Research* **1**, Morgan Kaufmann (February 1994), pp. 159-208.

[Lloyd87]
J.W. Lloyd, *Foundations of Logic Programming, Second Edition*, Springer Verlag (1987).

[Michalski80]
R.S. Michalski and R.L. Chilausky, "Learning by Being Told and Learning from Examples: An Experimental Comparison of the Two Methods of Knowledge Acquisition in the Context of Developing an Expert System for Soybean Disease Diagnosis," *Policy Analysis and Information Systems* **4**:2 (June 1980), pp. 125-160.

[Ourston91]
D. Ourston, *Using Explanation-Based and Empirical Methods in Theory Revision*, Ph.D. Thesis, University of Texas at Austin, Austin, TX (August 1991).

[Ourston90]
D. Ourston and R.J. Mooney, "Changing the Rules: A Comprehensive Approach to Theory Refinement," *Proceedings of the Eighth National Conference on Artificial Intelligence* (July 1990), pp. 815-820.

[Quinlan86]
J.R. Quinlan, "Induction of Decision Trees," *Machine Learning* **1**:1, Kluwer Academic (1986), pp. 81-106.

[Quinlan90]
J.R. Quinlan, "Learning Logical Definitions from Relations," *Machine Learning* **5**:3, Kluwer Academic (August 1990), pp. 239-266.

[Smith85]
R.G. Smith, H.A. Winston, T.M. Mitchell, and B.G. Buchanan, "Representation and Use of Explicit Justifications for Knowledge Base Refinements," *Proceedings of the Ninth International Joint Conference on Artificial Intelligence* (August 1985), pp. 673-680.

[Weiss91]
S.M. Weiss and C.A. Kulikowski, *Computer Systems That Learn: Classification and Prediction Methods from Statistics, Neural Nets, Machine Learning, and Expert Systems*, Morgan Kaufmann, San Mateo, CA (1991).