MITSUBISHI ELECTRIC RESEARCH LABORATORIES
http://www.merl.com

Learning Hierarchical Task Models By Demonstration

Andrew Garland and Neal Lesh

TR2003-01 February 2003

Abstract

Acquiring a domain-specific task model is an essential and notoriouslyand empirical results that measure the utility of possible annotations.

Submitted to IJCAI '03

This work may not be copied or reproduced in whole or in part for any commercial purpose. Permission to copy in whole or in part without payment of fee is granted for nonprofit educational and research purposes provided that all such whole or partial copies include the following: a notice that such copying is by permission of Mitsubishi Electric Research Laboratories, Inc.; an acknowledgment of the authors and individual contributions to the work; and all applicable portions of the copyright notice. Copying, reproduction, or republishing for any other purpose shall require a license with payment of fee to Mitsubishi Electric Research Laboratories, Inc. All rights reserved.

Copyright © Mitsubishi Electric Research Laboratories, Inc., 2003
201 Broadway, Cambridge, Massachusetts 02139

Submitted January 2003.

Learning Hierarchical Task Models By Demonstration

Content Areas: knowledge acquisition, machine learning

Abstract

Acquiring a domain-specific *task model* is an essential and notoriously challenging aspect of building knowledge-based systems. This paper presents machine learning techniques that ease this knowledge acquisition task. These techniques infer hierarchical models, including parameters for non-primitive actions, from partially-annotated demonstrations. Such task models can be used for plan recognition, intelligent tutoring, and other collaborative activities. Among the contributions of this work are a sound and complete learning algorithm and empirical results that measure the utility of possible annotations.

1 Introduction

Much work in AI aims to produce general algorithms that operate on declarative representations of domain-specific knowledge. However, encoding this knowledge is notoriously difficult and slow, and is dubbed the *knowledge acquisition bottleneck* to building knowledge-based systems. One approach to ease knowledge acquisition is to learn domain concepts from examples. In this work, we focus on the problem of acquiring *task models*, which are declarative representations of the actions that can be performed in a domain.

Past work on learning task models has investigated methods for learning from examples of sequences of actions executed to achieve a goal [Bauer, 1998, 1999; Lau *et al.*, 2000], and learning from information about the state of the world before and after actions are executed [Wang, 1995; van Lent and Laird, 1999; Angros Jr., 2000; Tecuci *et al.*, 1999]. Learned task models have been shown useful for a variety of tasks including plan recognition [Bauer, 1998, 1999], intelligent tutoring [Angros Jr., 2000], and action selection [van Lent and Laird, 1999].

We are interested in learning hierarchical task models because they allow learned concepts to be combined in novel ways. In this paper, we present and analyze techniques for inferring a hierarchical task model from examples provided by a domain expert. Each example is a demonstration of a goal-directed action sequence. Our techniques do not assume a demonstration contains state information; thus they are useful even when experimentation or observation of the state is not feasible. Because we learn hierarchical task models, one demonstration of a subtask applies to other tasks that include this subtask.

The primary contributions of this paper are:

- formalized notions of soundness and completeness for learning task models from examples. No past work has formalized desirable properties for task model learning.
- an implemented and provably sound and complete algorithm for learning task models from examples.
- methods for inducing both parameters of non-primitive actions and equality constraints between parameters, which we will refer to collectively as *propagators.*
- experiments in two domains that suggest an appropriate division of labor between the human expert and the learning algorithm.

Inducing propagators is a significant contribution because propagators are essential to develop an effective hierarchical task model, and are particularly difficult for people to specify. The next section gives examples of the role of propagators in task modeling and Section 3.1 shows the difficulty of learning them from examples. Past work on learning hierarchical task models [van Lent and Laird, 1999; Tecuci *et al.*, 1999] has not addressed learning propagators.

2 Task Model Learning

Informally, a task model learning algorithm must convert a series of demonstrations into a task model that accepts the set of action sequences that are consistent with the demonstrations. Our techniques leverage both syntactic and preference biases in order to infer sound and complete models.

2.1 Learning a hierarchical task model

Suppose one is trying to learn a model of making a pasta dinner based on the following two examples. In Example A, the chef performs five actions, labeled A1 to A5; in Example B, the chef performs actions B1 to B6. The indentations in each example constitute the *segmentations*, or groupings, that reflect the hierarchical nature of the task. For example, actions A3 and A4 collectively constitute preparing a sauce, as do B1 and B2. To simplify exposition, we modeled ingredients as part of the action types, e.g. AddGarlic(pot2) instead of Add(garlic1,pot2).

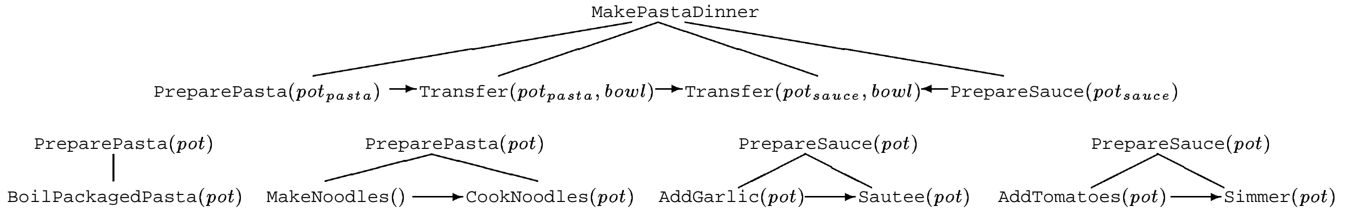

Figure 1: Sample task model consistent with Examples A and B.

```
Example A

MakePastaDinner
  PreparePasta
    A1. BoilPackagedPasta(pot1)
  A2. Transfer(pot1,bowl1)
  PrepareSauce
    A3. AddGarlic(pot2)
    A4. Sautee(pot2)
  A5. Transfer(pot2,bowl1)
```

```
Example B

MakePastaDinner
  PrepareSauce
    B1. AddTomatoes(pot3)
    B2. Simmer(pot3)
  PreparePasta
    B3. MakeNoodles()
    B4. CookNoodles(pot4)
  B5. Transfer(pot4,bowl2)
  B6. Transfer(pot3,bowl2)
```

These two examples imply many things about making a pasta dinner. For one, they indicate that there are four essential steps in making a pasta dinner: preparing the sauce, preparing the pasta, adding the pasta to the bowl, and adding the sauce to the bowl. The first two of these four steps can each be accomplished in two different ways. For instance, one can make either a tomato sauce or a garlic and oil sauce. The examples suggest that there are four possible combinations of sauce and pasta, although only two combinations were demonstrated.

In addition, the examples imply that there are various permissible orderings of the steps. It is possible to prepare the pasta before the sauce or vice versa. In contrast, the pasta is always added to the bowl before the sauce. Future examples, however, may show that ingredients may be added to the bowl in any order.

As will be seen later, these examples highlight the difficulties in learning propagators. Propagators constrain the values of parameters, such as making sure that the chef adds all of the ingredients to the same bowl. Similarly, the chef must cover the pasta with the contents of the pot containing the prepared sauce, not some other pot. However, different pots are used to prepare the sauce and the pasta. Such relationships are *non-local* because they involve constraining the parameters of actions in different segments.

Figure 1, described in the next section, presents a graphical depiction of a task model consistent with these examples.

2.2 Task model language

We learn hierarchical task models composed of actions and recipes. The set T of objects types (such as pots and bowls) is not learned but is treated as a given. Each learned model m is a pair $\langle \mathcal{A}, \mathcal{R} \rangle$, where $\mathcal{A}$ is a set of actions and $\mathcal{R}$ is a set of recipes.

An action is either a primitive action, which can be executed directly, or a non-primitive action, which is achieved indirectly by achieving other actions. Each action a in $\mathcal{A}$ is a pair $\langle name, [p_1, p_2, \ldots] \rangle$, where $name$ is unique identifier for a called an action type and each p_i is a parameter of a. In turn, each parameter p is a pair $\langle name, type \rangle$ where $name$ is unique among the parameters of a and $type$ is an object type. Note that this representation does not include any causal information, i.e. preconditions or effects.

Recipes are methods for decomposing non-primitive actions into subgoals. A recipe $r \in \mathcal{R}$ is composed of the four-tuple: $\langle name, achieves, [s_1, s_2, \ldots], C \rangle$, where $name$ is a unique identifier, $achieves \in \mathcal{A}$, each s_i is a step of r, and C is a set of constraints. In turn, each step s is a pair $\langle name, type \rangle$ where $name$ is unique among the steps of r and $type$ is an action type, i.e., the name of an action in $\mathcal{A}$. C imposes temporal partial orderings among the steps, as well as other logical relations among their parameters. In this paper, the only type of logical relation considered is equality. There may be several recipes for achieving a single action.

Figure 1 presents a graphical depiction of a task model that is consistent with Examples A and B. This model contains 5 recipes; for each recipe, the type of action that is achieved is the root of a one-level tree with the recipe steps as the leaves. Each temporal ordering constraint is represented by an arrow between two steps. Within each recipe, equality constraints are represented by using the same name for the parameters. For example, there is an equality constraint between the two parameters named pot_{pasta} in the recipe for achieving MakePastaDinner. (Note that steps are denoted by their type, whereas parameters are denoted by their name.)

We represent the demonstrations provided by the user as segments, which themselves can contain other segments. Formally, a *segment* $\mathcal{S}$ is a pair $\langle segmentType, [e_1, e_2, \ldots] \rangle$. Each e_i, called a *segment element*, is either a primitive action or a segment. Segmentations group together the actions that constitute an occurrence of a non-primitive act of type $segmentType$. An *annotated example* (also referred to as an *example*, below) is a segment which corresponds to an entire demonstration of a task by the user. Due to space constraints, segmentations are the only annotation we define precisely. Other annotations allow a domain expert to indicate that examples similar to the one being annotated are also acceptable, and can speed learning. We discuss several types of annotations in the Experiments section.

We now define what it means for a task model to *accept* an example. A given model m corresponds to a set of task networks: each network has an action from $\mathcal{A}$ at the top, with exactly one recipe applied to each non-primitive in the network, with an object assigned to each parameter in every action in the network, and with no constraint violations. Because the recipe steps are partially ordered, there are multiple total orderings of the primitive actions in a network. Each total ordering corresponds to exactly one segmentation, in which actions that achieve the same action are grouped together. Given this, accept(m, e) is defined as true for an example e iff e is the segmentation of a total ordering of a task network of m.

2.3 Soundness and completeness

This subsection describes the properties of soundness and completeness for task model learning. Roughly speaking, a sound and complete task model is the "intersection" of the set of task models that are consistent with the input examples. More precisely, a *sound* task model accepts only, but perhaps not all, examples that are accepted by every task model that is consistent with the input. A *complete* task model accepts all, and perhaps additional, examples that are accepted by every task model that is consistent with the input.

Let $\mathcal{E}$ be the set of possible annotated examples, and $\mathcal{M}$ be the set of possible task models. $\mathcal{M}$ may be partially ordered to reflect a preference order on its models (one will be introduces in Section 3.1). A *task model learning algorithm* Π takes a set of annotated examples $\bar{\mathcal{E}} \subset \mathcal{E}$ and returns a model $m \in \mathcal{M}$. Π is sound and complete if, for all $\bar{\mathcal{E}}$, the model $m = \Pi(\bar{\mathcal{E}})$ is sound and complete, as defined below:

- m is *consistent* with $\bar{\mathcal{E}}$ iff $\forall e \in \bar{\mathcal{E}}$, $\text{accept}(m, e)$.
- m is a *preferred consistent model* for $\bar{\mathcal{E}}$ if m is consistent with $\bar{\mathcal{E}}$ and $\forall m' \in \mathcal{M}$ that are consistent with $\bar{\mathcal{E}}$, m' is not ordered before m. Let $\text{PCM}(\bar{\mathcal{E}})$ be the set of all preferred consistent models for $\bar{\mathcal{E}}$.
- m is *sound* on $\bar{\mathcal{E}}$ iff for all $e \in \mathcal{E}$, $\text{accept}(m, e) \Rightarrow (\forall m' \in \text{PCM}(\bar{\mathcal{E}}), \text{accept}(m', e))$.
- m is *complete* on $\bar{\mathcal{E}}$ iff for all $e \in \mathcal{E}$, $\text{accept}(m, e) \Leftarrow (\forall m' \in \text{PCM}(\bar{\mathcal{E}}), \text{accept}(m', e))$.

Note that a preferred consistent model must be complete, but that a complete model may not be preferred.

3 Learning algorithm

This section details our algorithm for learning hierarchical task models from demonstrations. The focus is on inferring parameters of non-primitive actions and equality constraints between parameters, referred to collectively as *propagators*.

Figure 2 contains pseudo-code for our task model learning algorithm (called LEARNMODEL), which requires polynomial time. The first function called by LEARNMODEL, ALIGN, efficiently solves what we call the *alignment problem* by leveraging two syntactic biases on the hypothesis space of task models.

The alignment problem is determining which segments, possibly in different examples, correspond to the same recipe, and which segment elements correspond to the same recipe step. For example, suppose we know action sequences a_1, a_2, a_3 and a_3, a_2, a_1 both achieve the same non-primitive. These examples might both correspond to the same recipe, with no ordering constraints, or might correspond to two different, totally-ordered recipes. Similarly, there can be ambiguity about which segment elements correspond to the same recipe step if segments contain multiple steps of the same type. We address the alignment problem by introducing the following syntactic biases:

Unique steps assumption: if two recipes achieve the same non-primitive, they have different sets of step types or a different number of steps of some type.

Multiple steps assumption: if a recipe contains multiple steps of the same type, those steps are totally ordered.

LEARNMODEL $(\bar{\mathcal{E}}) \equiv$
  $m_1 \leftarrow$ ALIGN$(\bar{\mathcal{E}})$
  $m_2 \leftarrow$ INDUCEORDERING$(m_1, \bar{\mathcal{E}})$
  return INDUCEPROPAGATORS$(m_2, \bar{\mathcal{E}})$

Figure 2: Pseudo-code to learn a task model

These assumptions hold in the domains we have examined; in other domains, other biases or heuristics may be needed to alleviate the search problem faced by ALIGN. The other components of our learning algorithm do not depend on these assumptions.

ALIGN groups all segments together that have the same segment type and whose segment elements have the same set of segment types (counting repeats). ALIGN constructs a recipe for each group, with a step for each segment element. It then constructs a mapping from each segment in a group to the recipe that was created for that group. (The mappings created by ALIGN are used by subsequent functions in LEARNMODEL). For every segment, ALIGN maps the ith segment element it contains of a given type to the ith recipe step of that type in the recipe that the segment is mapped to.

Next, our algorithm determines ordering constraints between steps. The INDUCEORDERINGS function adds a constraint that orders step s_i before step s_j unless there is a segment that contains elements e_i and e_j such that e_i is mapped to s_i and e_j is mapped to s_j and e_j occurs before e_i.

The following theorem states that the first two functions of LEARNMODEL produce a task model that is correct, except that it contains no propagators.

Theorem 1: If there exists a sound and complete model for example set $\bar{\mathcal{E}}$, then there exists a sound and complete model m such that INDUCEORDERINGS(ALIGN($\bar{\mathcal{E}}$),$\bar{\mathcal{E}}$) and m differ only in their propagators.

Proof sketch: Let m' be a sound and complete model on $\bar{\mathcal{E}}$. For every segment s in every example in $\bar{\mathcal{E}}$, there must be a recipe with the same distribution of step types as segment element types in s, or else m' could not accept that example. Conversely, any recipe in m' whose distribution of step types does not match any segment's types can be removed from m' without changing the examples accepted by m', or else m' would not be sound. Let m be the result of removing such extraneous recipes from m'.

Given the unique steps assumption, m must have the same recipes as ALIGN, ignoring propagators and ordering constraints. Given the multiple steps assumption, there is only one possible mapping between segment elements and recipe steps. If m allows a step s_i to be ordered before a step s_j, then s_i must maps to a segment element that appears before the element that s_j maps to in some segment, or else m will not be sound. Thus, m and the model produced by INDUCEORDERINGS will allow the same orderings.□

3.1 Inducing propagators

A challenge of learning hierarchical task models is that they must enforce equality relationships that cross the boundaries of many actions and recipes. For example, a task model to describe changing a flat tire must ensure that the wheel is the same, but different tires will be used. Propagators collectively enforce such non-local equalities.

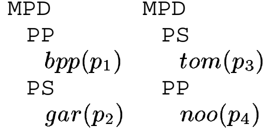

Figure 3: Two examples

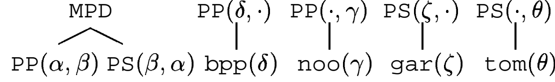

Figure 4: A counter-intuitive consistent task model

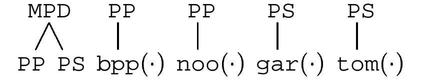

Figure 5: The preferred task model

A problem arises when learning sound task models that include propagators, however. Figures 3 to 5 illustrate this problem. Figure 3 contains simplified versions of Examples A and B, using abbreviated action names (e.g. *bpp* instead of BoilPackagedPasta) and omitting some actions; Figures 4 and 5 show task models consistent with the examples. In these figures, all parameters are pots and a dot indicates a parameter that is not constrained.

Consider the decision of whether to add propagators to enforce an equality constraint between the parameters of two recipe steps in the task model. For any set of annotated examples $\bar{\mathcal{E}}$, there are three situations to consider:

1. *contradicting evidence* exists in $\bar{\mathcal{E}}$, i.e., there is an example e in $\bar{\mathcal{E}}$ where the parameters are unequal. In Figure 3, the left-hand example provides contradicting evidence between the pot parameters of the steps of type *bpp* and gar.
2. *only supporting evidence* exists in $\bar{\mathcal{E}}$, i.e. there are some examples where the steps' parameters are equal, and no contradicting evidence. This case holds for the bowl parameters in Examples A and B.
3. *no relevant evidence* exists in $\bar{\mathcal{E}}$, i.e., there is no example that contains both steps. In Figure 3, the pot parameters for the steps of type *bpp* and tom fall into this category. The examples imply that doing *bpp* and then tom will achieve MPD, but no example contains both steps.

In the first case, clearly the task model should not enforce an equality between the parameters. In the second case, it is possible that all the supporting evidence has been coincidental, but until contradictory evidence is seen, a sound model must only accept examples in which the two steps' parameters have the same value.

The third case is not as straightforward. As shown by Figure 4, there exist consistent task models that constrain the parameters of steps to be equal if there has been no evidence given about their relationship. The model in Figure 4 is consistent with all observed data because the elaborate propagators it contains only effect the *unobserved* combinations of preparing sauce and pasta.

More generally, by the definition of soundness, every sound model must also constrain parameters with no relevant evidence to be equal. Thus, a sound and complete learning algorithm must treat the *no relevant evidence* case the same as the *only supporting evidence* case.

The problem with such models is that they postulate elaborate constraints that are not suggested by any example. To avoid adding any such counter-intuitive constraints, the learning algorithm must be given an example for every pair of parameters that are *unrelated* to each other.

To address this problem, we propose the following bias:

Propagators with support preference bias: We prefer any model in which every propagator has *only supporting evidence* in $\bar{\mathcal{E}}$ to all models that contain a propagator for which *no relevant evidence* exists in $\bar{\mathcal{E}}$.

The effect of the bias is shown in Figures 4 and 5. In Figure 4, no propagators have supporting evidence because no parameter values are equal in the two input examples. Thus, the preferred task model in Figure 5 is sound and complete.

3.2 Propagator induction algorithm

Figure 6 shows pseudo-code for an algorithm for learning propagators. A data structure that facilitates the computation of propagators is a $path$. A path "starts" at a parameter of a primitive action and "follows" a possibly empty sequence of recipe steps. Given a path p, PARAMETER(p) returns the parameter at the start of the path; also, if p has a non-empty sequence of steps, STEP(p) returns the last recipe step and RECIPE(p) returns the recipe that contains STEP(p).

```
INDUCEPROPAGATORS (m,Ē) ≡
  forall R in ALLRECIPES(m)
    ADDCONSTRAINTS(R,Ē)
ADDCONSTRAINTS (R,Ē) ≡
  P ← PATHSTORECIPE(R,Ē)
  forall sets {p,p'} in PATHPAIRINGS(P,Ē)
    name ← PROPAGATENAME(p,null)
    name' ← PROPAGATENAME(p',null)
    add a constraint between parameter name of STEP(p)
        and parameter name' of STEP(p')
PROPAGATENAME (p, purposeSlot) ≡
  tail ←TAIL(p)
  if tail has no steps
    then slotName ← NAME(PARAMETER(p))
    else
      slotName ← GENSYM()
      PROPAGATENAME(tail, slotName)
  if purposeSlot ≠ null
    R ← RECIPE(p)
    add a parameter named purposeSlot of type
        TYPE(PARAMETER(p)) to PURPOSE(R)
    add a constraint between purposeSlot of PURPOSE(R)
        and parameter slotName of STEP(p) to R
  return slotName
PATHPAIRINGS (P,Ē) ≡
  L ← ∅
  forall p, p' in P such that p ≠ p'
    if PARAMETER(p) and PARAMETER(p')
        have only supporting evidence in Ē
        or (the preference bias is not in effect
            and no relevant evidence exists in Ē)
      then L ← L ∪ {{p, p'}}
  return L
```

Figure 6: Pseudo-code to infer propagators

The algorithm works by considering all pairs of paths that end at the same recipe $\mathcal{R}$. If the parameters at the start of these paths should always be constrained to be equal (the criteria for this depends on the preference bias), then a set of propagators are added to the task model to make sure this will be the case. The propagators are added in a top down fashion, first with a constraint on $\mathcal{R}$, and then recursively adding parameters to non-primitives and constraints to recipes that achieve them.

Theorem 2: Given a set $\bar{\mathcal{E}}$, and a task model m without any propagators such that there exists a model m' that is sound and complete on $\bar{\mathcal{E}}$, and that m and m' differ only in their propagators, then INDUCEPROPAGATORS$(m, \bar{\mathcal{E}})$ will return a sound and complete model.

Proof sketch: The role of propagators is to enforce equality among the parameters of primitive actions that must be equal, based on the annotated examples. Since equality is a binary, transitive relationship, it suffices to consider parameters on a pair-wise basis. If any parameters have been unequal in any of the annotated examples, then our algorithm will not make them equal. This is appropriate since this example implies that a consistent model should not force them to be equal. Otherwise, without a preference bias, our algorithm will force the parameters to be equal which is appropriate since there exists a preferred, consistent model which forces the parameters to be equal. If we use the propagators with support preference bias, then our algorithm will not force the pair of parameters to be equal which is appropriate since any model that enforces equality will contain unsupported propagators. □

It follows as a corollary of Theorems 1 and 2 that LEARNMODEL is sound and complete.

4 Implementation and empirical results

The algorithm described in the previous section is a simplified version of the one we have implemented. Our implementation accepts many kinds of annotations other than segmentations. In addition, it can learn a wider class of task models than described earlier, where recipe steps can be labeled as optional. (Space limitations prevent technical discussion of optional steps, including its effect on the alignment problem, in this paper.) Furthermore, our implementation is incremental and while the INDUCEPROPAGATORS algorithm we presented produces an inordinate number of propagators, our implementation re-uses propagators when possible.

An example of the annotations our system accepts is that the user can specify that two actions in an example could have been performed in either order. This directly provides the same information that a second example might have provided. Another important annotation is that the user can directly specify that two parameters do not have to be equal. Other possible annotations are non-primitive parameters, recipe names, step names, or step optionality.

The goal of our experiments is to better understand the tradeoff between how many annotations the expert provides in each example and how many examples must be provided. In order to do so, we simulate a human expert that provides varying types of annotations. This approach focuses the results on this tradeoff rather than the best way to elicit annotations from the expert.

We based our experiments on two manually created task models. The first models part of a sophisticated tool for building graphical user interfaces, called Symbol Editor. The model was constructed in the process of developing a collaborative agent to assist novice users. The model contains 29 recipes, 67 recipe steps, 36 primitive acts, and 29 non-primitive acts. A typical example contains over 100 primitive actions. The second test model was a cooking world model designed specifically to develop and test the techniques presented in this paper. The model contains 8 recipes, 19 recipe steps, 13 primitive acts, and 4 non-primitive acts. An example typically contains about 10 primitive actions. Both models have recursive recipes.

Segmentations and non-primitive action names (i.e., segment types) are always provided by the simulated expert, but we varied whether the other annotations were provided. For each combination of annotations, we use the known task models to generate a corpus of annotated examples (500 for the symbol editor and 1000 for cooking). Then we ran the learning algorithm on all examples and hand-verified that the produced task model (called the target task model below) was semantically equivalent to the original task model.

For each learning trial, input examples were drawn at random (without replacement) from the corpus. After each example, we determine if the algorithm has produced a task model that accepts the same sequences as the target task model. Also, for each example, we determine if it was *useful*, i.e. if it contained any new information that was not implied by the previous examples, by seeing if the algorithm's internal data structures were altered.

We ran all possible combinations of annotation types, and report a subset in Table 1. In the table, O indicates that all ordering annotations are given, I indicates that all inequality annotations are given, and P indicates that all non-primitive parameters are given. Unlike other runs, the annotations for "All" include recipe and step names. The reported values are averaged over randomized sequences of examples — 100 trials for each domain. All columns (other than the one labeled "Useless") report statistics about the distribution of the number of useful examples required to match the target task model.

One conclusion to be drawn from Table 1 is that non-primitive parameters are the single-most useful kind of annotation that can be provided. This is unsurprising since it frees the algorithm from trying to learn the most complicated relationships in the data. The main surprise is that providing inequality annotations significantly reduces the number of useful examples, whether or not non-primitive parameters are provided (compare rows "I" and "None" as well as rows "PI" and "P"). This is interesting because it seems likely that a human expert can easily indicate when apparent equalities in the example are coincidental.

Table 1 also shows that learning is strongly influenced by the order in which examples are processed. This is reflected both by the minimum number of useful examples for any trial (the "min" column) and the average number of useless examples per trial (the "useless" column). We suspect that a human expert would present examples with high utility.

The column labeled "Error (n)" in Table 1 shows the error rate after n useful examples have been seen. The error rate is measured as the fraction of the total information that remains

Additional Annotations	Cooking						Symbol Editor					
	Avg.	Dev.	Min.	Max.	Useless	Error (5)	Avg.	Dev.	Min.	Max.	Useless	Error (1)
All	5.27	1.43	3	10	8.02	2.9%	1.71	0.57	1	3	0.05	1.9%
PIO	6.56	1.43	3	10	10.67	4.7%	1.71	0.57	1	3	0.05	1.9%
PI	7.33	1.56	4	11	16.46	5.2%	2.90	0.64	2	4	0.75	2.2%
PO	10.99	2.02	5	15	16.51	11.6%	2.94	0.63	2	5	0.28	3.2%
P	11.31	2.10	5	16	19.04	11.6%	3.55	0.76	2	6	0.60	3.4%
IO	14.64	3.38	6	22	54.43	6.3%	3.84	1.11	2	7	0.18	2.1%
I	15.04	3.39	6	22	54.08	6.6%	4.38	1.22	2	7	0.22	2.3%
O	27.96	5.40	15	46	183.00	13.2%	8.77	1.90	5	15	1.97	4.6%
None	28.09	5.46	15	46	182.87	13.4%	8.84	1.84	5	15	1.91	4.8%

Table 1: The kind of annotations provided influences the number of examples needed to learn task models.

to be learned, i.e. how much the internal representations of the current task model and the target model differ. The table shows that even when it takes many examples to learn the correct model, e.g., when no extra annotations are given, the techniques quickly learn a model which is close to the correct model.

5 Related research and Conclusion

The hierarchical nature of our input data makes learning task models an unusual concept learning or inductive logic programming problem. The learned concepts, i.e. task models, are composed of many smaller concepts, i.e. actions and recipes. Each example can provide information that generalizes each concept in isolation, as well as information about how the concepts interrelate. Our techniques can be seen as specialized and efficient solutions that leverage the hierarchical characteristics of this concept learning problem.

Bauer (1998; 1999) presents techniques for acquiring non-hierarchical task models from unannotated examples for the purpose of plan recognition. Bauer introduces heuristics for solving what we refer to as the alignment problem. (In contrast, we side-step the problem by restricting the task model language.) We extend upon Bauer's work to handle hierarchical task models and optional steps. Additionally, we introduce the notions of soundness and completeness for task model learning and show our algorithm has these properties.

Tecuci *et al.* [1999] present techniques for producing hierarchical if-then task reduction rules by demonstration and discussion from a human expert. The rules are intended to be used by knowledge-based agents that assist people in generating plans. In their system, the expert provides a problem-solving episode from which the system infers an initial task reduction rule, which is then refined through an iterative process in which the human expert critiques attempts by the system to solve problems using this rule. Tecuci *et al.* have not presented formal analysis of their algorithms, specifically addressed the problem of inferring parameters for learned actions, or conducted experimental exploration of the division of responsibility between the user and learning algorithms.

Other research efforts have addressed aspects of the task model learning problem not addressed in this paper. Angros Jr. [2000] presents techniques that learn recipes that contain causal links, to be used for intelligent tutoring systems, through both demonstration and automated experimentation in a simulated environment. Lau *et al.* [2000], in one of the few formal approaches to learning macros, use a version space algebra to learn repetitive tasks in a text-editing domain. Gil *et al.* [Gil and Melz, 1996; Kim and Gil, 2000] have focused on developing tools and scripts to assist people in editing and elaborating task models, including techniques for detecting redundancies and inconsistencies in the knowledge base, and making suggestions to users about what knowledge to add next.

In conclusion, this paper presented the first formal definitions of soundness and completeness of task model learning, and a sound and complete algorithm for learning task models from partially-annotated examples. An important and novel aspect of our algorithm is that it learns hierarchical task models, including propagators. Finally, we conducted an empirical study that suggested human experts can significantly speed learning simply by noting when apparent equalities are coincidental.

References

Richard Angros Jr. *Learning What to Instruct: Acquiring Knowledge from Demonstrations and and Focussed Experimentation*. PhD thesis, University of Southern California, 2000.

Mathias Bauer. Acquisition of Abstract Plan Descriptions for Plan Recognition. In *Proc. 15th Nat. Conf. AI*, pages 936–941, 1998.

Mathias Bauer. From Interaction Data to Plan Libraries: A Clustering Approach. In *Proc. 16th Int. Joint Conf. on AI*, pages 962–967, 1999.

Y. Gil and E. Melz. Explicit representations of problem-solving strategies to support knowledge acquisition. In *Proc. 13th Nat. Conf. AI*, pages 469–476, 1996.

J. Kim and Y. Gil. Aquiring problem-solving knowledge from end users: Putting interdependency models to the test. In *Proc. 17th Nat. Conf. AI*, pages 223–229, 2000.

Tessa Lau, Pedro Domingos, and Daniel Weld. Version space algebra and its application to programming by demonstration. In *Proc. 17th Int. Conf. on Machine Learning*, pages 527–534, 2000.

G. Tecuci, M. Boicu, K. Wright, S. Lee, D. Marcu, and M Bowman. An integrated shell and methodology for rapid development of knowledge-based agents. In *Proc. 16th Nat. Conf. AI*, pages 250–257, 1999.

Michael van Lent and John Laird. Learning hierarchical performance knowledge by observation. In *Proc. 16th Int. Conf. on Machine Learning*, pages 229–238. Morgan Kaufmann, San Francisco, CA, 1999.

Xuemei Wang. Learning by observation and practice: an incremental approach for planning operator acquisition. In *Proc. 12th Int. Conf. on Machine Learning*, pages 549–557, 1995.