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

Learning Hierarchical Task Models By Demonstration

Andrew Garland
Neal Lesh

TR-2002-04 January 2002

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 which are built into an interface that eases 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.

Submitted to AAAI-02

Submitted January 2002.

Learning Hierarchical Task Models By Demonstration

Andrew Garland and Neal Lesh
Mitsubishi Electric Research Laboratories
{garland,lesh}@merl.com

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 which are built into an interface that eases 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, Domingos, & Weld, 2000), and learning from information about the state of the world before and after actions are executed (Wang, 1995; van Lent & Laird, 1999; Angros Jr., 2000; Tecuci *et al.*, 1999). In the latter approach, the learning system is often allowed to experiment by executing actions and observing their effects. 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 & Laird, 1999).

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 that is *partially annotated*. 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 expert can annotate a demonstration, for example, to indicate that two actions can occur in either order. This work is part of an ongoing project to develop user interface agents (*references omitted for review*).

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. Consider, for example, trying to learn a task model that describes how to repair an automobile from repeated demonstrations. In this case, the system must learn that the engine removed from the car must be the same engine that is returned to it. In contrast, a bolt that is unscrewed from a place is not normally screwed back into that place later. Past work on learning hierarchical task models (van Lent & 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. Each demonstration shows one acceptable way to perform a task and uses annotations to indicate additional related acceptable sequences. For example, if the sequence $[a, b, c]$ is demonstrated and b is annotated as optional, then $[a, c]$ is also an acceptable way to perform the task. Our techniques leverage both restrictive and preference biases in order to infer models with desirable properties.

2.1 Task model and annotation languages

We learn hierarchical task models composed of actions and recipes. Actions are either primitive actions, which can

nonprimitive act PreparePasta
  parameter type Pasta **name** pasta
recipe PastaRecipe **achieves** PreparePasta
  steps **type** BoilWater **name** boil
  type MakePasta **name** make,
  optional type GetItem **name** get
  constraints **achieves**.pasta = make.pasta
  get.item = make.pasta, boil.liquid = make.water
  boil **precedes** make, get **precedes** make
primitive act GetItem
  parameter type Item **name** item

Figure 1: Sample textual representation of a portion of a task model for a simple cooking domain. Keywords are in bold (**achieves** refers to the non-primitive action that is the purpose of a recipe). Parameters and steps have a name as well as a type to allow for unambiguous references in constraints.

be executed directly, or non-primitive actions, which are achieved indirectly by achieving other actions. Each action has a type; each action type is associated with a set of parameters, but we do not assume that any causal information, i.e. preconditions and effects, will be provided.

Recipes are methods for decomposing non-primitive actions into subgoals. There may be several different recipes for achieving a single action. Each recipe defines a set of steps that can be performed to achieve a non-primitive action and constraints that impose temporal partial orderings on its steps, as well as other logical relations among their parameters. In this paper, the only type of logical relation considered is equality. All steps are assumed to be required unless they are labelled as optional.

Annotations allow a domain expert to indicate that examples similar to the one being annotated are also acceptable. For the class of hierarchical models we are learning, an annotated example e is a five-tuple: $\langle \hat{e}, \mathcal{S}, optional, unordered, unequal \rangle$:

- $\hat{e}$ is the sequence of primitive actions $[p_1, \ldots, p_k]$ that constitute the unannotated example.
- $\mathcal{S}$ is a segment; a segment is a pair $\langle segmentType, [s_1, \ldots, s_n] \rangle$. Each s_i, called a *segment element*, is either a primitive action or a segment. Segment annotations group together the actions that constitute an occurrence of a non-primitive act of type $segmentType$.
- $optional$ is a partial boolean function on segment elements. When $optional(s_i)$ is true, the example derived from e by removing s_i is also acceptable.
- $unordered$ is a partial boolean function on pairs of elements in the same segment. When $unordered(s_i, s_j)$ is true, the example derived from e by switching the order of appearance of s_i and s_j is also acceptable.
- $unequal$ is a partial boolean function on pairs of action parameters. When $unequal(p_1, p_2)$ is true, there exists another acceptable example, identical to e except for parameter values, where $p_1 \neq p_2$.

For the algorithms in this paper, the segments must be provided by the domain expert. The other annotations are not required, but will speed learning.

2.2 Soundness and completeness

We now describe the properties of soundness and completeness for a task model learning algorithm. Roughly speaking, a sound and complete task model is the "intersection" of the set of task models that are consistent with the input examples. A task model learning algorithm is sound and complete if it always produces a sound and complete model.

A sound and complete task model can be described more clearly, though still informally, in terms of the set of action sequences that a model will *accept* (or, equivalently, produce). 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.

Making these ideas precise requires some notation: $\mathcal{P}$ is the set of primitive actions, $\mathcal{E}$ is the set of possible annotated examples, and $\mathcal{M}$ is the set of possible task models. $\mathcal{M}$ may be partially ordered to reflect a preference order on its models. Let $\mathcal{P}^*$ be the set of all finite sequences of the primitives in $\mathcal{P}$. For any annotated example e, let $\hat{e} \in \mathcal{P}^*$ be the unannotated action sequence.

For any combination of task model and annotation languages, we assume there exists a function *accept*(m, p^*) that returns true if and only if p^* is an action sequence that m could produce. Similarly, *accept*(m, e) returns true if and only if annotated example e could be produced by m.

A *task model learning algorithm* $\mathcal{A}$ takes a set of annotated examples $\bar{\mathcal{E}} \subset \mathcal{E}$ and returns a model $m \in \mathcal{M}$. $\mathcal{A}$ is sound and complete if, for all $\bar{\mathcal{E}}$, the model $m = \mathcal{A}(\bar{\mathcal{E}})$ is sound and complete, as defined below:

- m is *consistent* with $\bar{\mathcal{E}}$ iff $\forall e \in \bar{\mathcal{E}}$, 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 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 $p^* \in \mathcal{P}^*$, accept$(m, p^*) \Rightarrow (\forall m' \in \text{PCM}(\bar{\mathcal{E}}), \text{accept}(m', p^*))$.
- m is *complete* on $\bar{\mathcal{E}}$ iff for all $p^* \in \mathcal{P}^*$, accept$(m, p^*) \Leftarrow (\forall m' \in \text{PCM}(\bar{\mathcal{E}}), \text{accept}(m', p^*))$.

Figure 2 shows an example of task model learning. The learned model learned shows a key benefit of hierarchical models: it accepts many action sequences that have not been seen, such as $[a(1), e(2), g(3), h(3), d(1, 2)]$. This model is complete; it is sound when models are ordered using the preference bias introduced in the next section.

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*. We present pseudo-code for the method that induces propagators, and show that it is sound and complete. Other important parts of the learning algorithm will be described, but space limitations prevent including code or proofs pertaining to them.

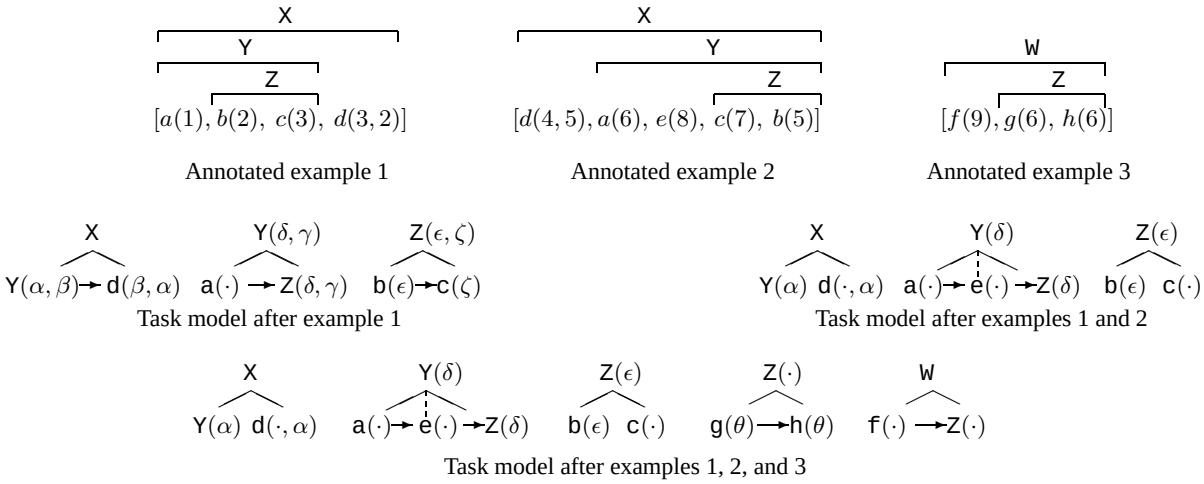

Figure 2: Example of task model learning. The unannotated examples are sequences of primitive actions, shown in lower-case italics, such as $[a(1), b(2), c(3), d(4)]$. The action $a(2)$ represents an action of type a with parameter 2. The only annotations in these examples are segmentations which cluster actions into groups that achieve a single non-primitive action, shown in upper-case letters, such as X, Y, Z. The recipes contain both primitive and non-primitive action types, shown by lower and upper case letters, respectively. Dotted lines indicate optional steps. Arrows indicate ordering constraints. A Greek letter represents two or more parameters of actions that are bound to be equal by a recipe. A "·" indicates that an action has a parameter that is not bound to any other parameter in that recipe.

Figure 3 contains pseudo-code for our task model learning algorithm (called LEARNMODEL), which requires polynomial time. The first function called by LEARNMODEL, ALIGN, efficiently solves a key search problem by leveraging two restrictive biases. The problem is to determine which segments, possibly in different examples, correspond to the same recipe, and which segment elements correspond to the same recipe step. Solving this problem is essential in order to perform useful generalization. ALIGN is efficient because of the following assumptions that restrict the class of task models our algorithm will learn:

Disjoint steps assumption: for any two recipes that achieve an action of the same type, the sets of the types of their required steps will be disjoint.

Step type assumption: if any recipe contains multiple steps of the same type, they will be totally ordered and only the last might be optional.

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. For example, if one assumes that a domain expert always provides recipe or step names when there is ambiguity, these assumptions are not needed. The other pieces of our learning algorithm do not depend on these assumptions.

ALIGN constructs a model m that contains non-primitive actions without parameters and recipes without constraints or optional steps. Given our restrictive assumptions, only one such task model exists for any set of annotated examples, and it is easily computed. During this construction process, ALIGN creates two maps, one from segments to model recipes and one from segment elements to recipe steps, which are used by the remaining functions called by LEARNMODEL.

Next, our algorithm determines step optionality and ordering constraints between steps. The INDUCEOPTIONAL function marks step s in recipe r as optional if any segment element that is mapped to s is marked optional or if any segment that is mapped to r contains no element that is mapped to s. The INDUCEORDERING 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 either e_j occurs before e_i or the annotations indicate that this ordering was possible.

LEARNMODEL $(\bar{\mathcal{E}}) \equiv$
  $m_0 \leftarrow$ ALIGN$(\bar{\mathcal{E}})$
  $m_1 \leftarrow$ INDUCEOPTIONAL$(m_0, \bar{\mathcal{E}})$
  $m_2 \leftarrow$ INDUCEORDERING$(m_1, \bar{\mathcal{E}})$
  return INDUCEPROPAGATORS$(m_2, \bar{\mathcal{E}})$

Figure 3: Pseudo-code to learn a task model

3.1 Inducing propagators

We are interested in learning hierarchical task models because they allow learned concepts to be combined in novel ways. For example, any example that demonstrates a way to remove an engine from a car also implicitly shows an additional way to accomplish other tasks that necessitate removing a car engine.

On the other hand, a challenge of learning hierarchical task models is that they must enforce equality relationships that cross the boundaries of many actions and recipes, i.e. that are not local to any particular recipe. For example, a task model to describe changing flat tires must ensure that the car is always the same, but different tires will be used. Propagators, i.e., parameters of non-primitive actions and equality constraints (in recipes) between parameters, collectively enforce such non-local equalities.

A problem arises when learning sound task models that include propagators, however. Consider the decision of whether to add a constraint between the parameters of two

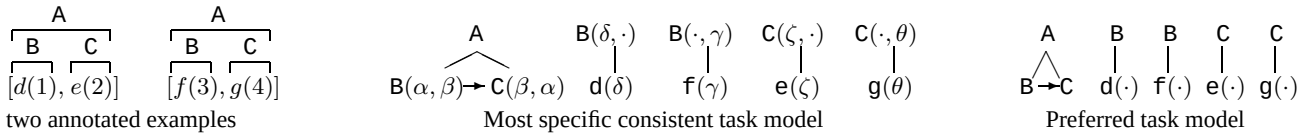

Figure 4: Motivation for propagators with support preference bias: the task models differ in that only the first contains propagators that force d and g's parameters and e and f's parameters to be equal. Both models are consistent because they each accept both examples. Without the preference bias, however, only the more elaborate model is sound because the simpler model accepts examples (e.g., $[d(1), g(2)]$) which are not accepted by all consistent models. With the bias, the simpler model is sound because it is preferred to the elaborate model.

recipe steps in the task model. For any set of annotated examples, there are three situations to consider:

1. *negative evidence* for the constraint exists, i.e., there is an example showing the parameters are unequal. In Figure 4, example $[d(1), e(2)]$ provides negative evidence between the parameters of step d (in the first recipe to accomplish B) and step e (in the first recipe to accomplish C).
2. *only positive evidence* exists for the constraint, i.e. there are some examples where the steps' parameters are equal, and no negative evidence. This case holds for steps g and h in the example $[g(6), h(6)]$ that achieved Z in Figure 2.
3. *no relevant evidence* exists, i.e., there is no example that contains both steps. In Figure 4, steps d and g fall into this category since the examples imply that doing d and then g will achieve A, 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 positive evidence has been coincidental, but until negative evidence is seen, a sound model must only accept examples in which the two steps' parameters are equal.

The third case, however, is interesting. A sound and complete learning algorithm must treat the *no relevant evidence* case the same as the *only positive evidence* case. Recall that in order to be sound, an algorithm must produce a model that only accepts action sequences that are accepted by all consistent models. So, if any model is consistent with the input examples and has an equality constraint between two parameters, the learning algorithm must produce a model that enforces that constraint. As shown in 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 problem is that such models are counter-intuitive, because they postulate elaborate constraints that are not suggested by any example. Further, to remove each unnecessary constraint, the learning algorithm must see an example that contains a pair of steps that are *unrelated* to each other.

To address this problem, we propose a bias against models with propagators that are not supported by positive evidence. In this approach, we assume a constraint between parameters does not exist until there is evidence that it does, and retract a constraint given evidence that it does not exist. This bias can be stated as the following:

Propagators with support preference bias: A model m_i is preferred to model m_j given annotated examples $\bar{\mathcal{E}}$ iff all of m_i's propagators are supported by only positive evidence in $\bar{\mathcal{E}}$ and m_j has propagators that are not supported by any example in $\bar{\mathcal{E}}$.

The effect of the bias is shown in Figure 4. In this figure, no propagators are supported by positive evidence because no parameter values are equal in the two demonstrations. Note that an non-preferred model may become preferred as more examples are seen. In Figure 4, if we saw $[d(1), g(1)]$ and $[f(1), e(1)]$ then all the propagators in the more elaborate task model would be supported and so it would be preferred.

3.2 Propagator induction algorithm

Figure 5 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.[1] 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).

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.

The following theorem states that our algorithm will produce a sound and complete model by adding propagators to its input model.

Theorem: 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

[1] Think of an annotated example as a tree, with the top-level segment as the root, segment elements as the interior nodes, and parameters of primitive actions as the leaves. A path starts at a leaf and goes up all or part of the tree.

```
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
        never been negatively related in ℰ̄
        and either p and p' have been positively related in ℰ̄
            or the preference bias is not in effect
      then ℒ ← ℒ ∪ {{p, p'}}
  return ℒ
```

Figure 5: Pseudo-code to infer propagators

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 does enforces equality will contain unsupported propagators.

It follows that if the alignment and other induction components of our algorithm are correct, then LEARNMODEL is sound and complete.

4 Implementation and empirical results

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.

The algorithm described in the previous section is a simplified version of the one we have implemented. Our implementation is incremental and accepts a wider class of annotations, including explicitly providing non-primitive parameters, recipe names or step names. Also, while the INDUCEPROPAGATOR algorithm we presented produces an inordinate number of propagators, our implementation re-uses propagators when possible. Our system allows task models to be edited in order to give semantically meaningful names to recipes and steps.

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.[2] Additionally, we determine if each example was *useful*, i.e. if it contained any new information that was not implied by the previous example, 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.

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 required 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 required examples for any

[2] Since both models are produced by the same algorithm, it is sufficient to see if a one-to-one mapping between actions and recipes in the two models exists.

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.

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 labelled "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 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

Bauer (1998; 1999) presents techniques for acquiring non-hierarchical task models from unannotated examples for the purpose of plan recognition. Since the task model is used primarily for recognition, Bauer's algorithm learns only the required steps to accomplish each top-level goal. 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.) Since our task models are intended to support collaboration and discussion of tasks, we found it important to extend 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, Domingos, & Weld (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 & Melz, 1996; Kim & 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

Angros Jr., R. 2000. *Learning What to Instruct: Acquiring Knowledge from Demonstrations and and Focussed Experimentation*. Ph.D. Dissertation, University of Southern California.

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

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

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

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

Lau, T.; Domingos, P.; and Weld, D. 2000. Version space algebra and its application to programming by demonstration. In *Proc. 17th Int. Conf. on Machine Learning*, 527–534.

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

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

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