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

Learning Hierarchical Task Models by Demonstration (subsumed by TR2002-04)

Andrew Garland and Neal Lesh

TR2001-03 January 2001

Abstract

Most previous work on learning task models, a special case of the well-known knowledge acquisition bottleneck, has dealt with non-hierarchical models. We present and analyze techniques for inferring a hierarchical task model from partially-annotated examples of task-solving behavior. We show our algorithm has desirable formal properties and that both restrictive and preference biases are useful for generating effective models. Finally, we describe experiments that explore the appropriate division of labor between the learning algorithm and the person who provides the annotated examples.

Submitted to IJCAI-01, January 2001.

Learning Hierarchical Task Models By Demonstration

Content Areas: machine learning, knowledge acquisition
Tracking Number: 580

Abstract

Most previous work on learning task models, a special case of the well-known knowledge acquisition bottleneck, has dealt with non-hierarchical models. We present and analyze techniques for inferring a hierarchical task model from partially-annotated examples of task-solving behavior. We show our algorithm has desirable formal properties and that both restrictive and preference biases are useful for generating effective models. Finally, we describe experiments that explore the appropriate division of labor between the learning algorithm and the person who provides the annotated examples.

1 Introduction

Much work in artificial intelligence is based on using general algorithms that operate on declarative representations of domain-specific knowledge. This approach gives rise to the notorious *knowledge acquisition bottleneck*: developing an effective declarative model is a significant obstacle to applying general AI methods to a domain.

We focus on the problem of acquiring *task models*, i.e., declarative representations of how to decompose and accomplish goals and subgoals. Task models are used in many areas of AI including planning, intelligent tutoring, plan recognition, and decision theory. Most previous work on learning task models has dealt only with "flat" models that decompose each goal directly into primitive actions (Bauer, 1998, 1999; Angros Jr., 2000). That work, however, ignores one of the most difficult aspects of learning task models; namely, deciding how to divide tasks into subtasks, which involves choosing the best abstractions to represent intermediate goals.

Our interest in hierarchical task models is motivated, in part, by our involvement in developing collaborative interface agents (*references omitted for review*). The choice of intermediate goals is especially important in this context because the agent must be able to discuss how to accomplish tasks in a way that is intuitive to the user. Hierarchical task models are also important in planning applications (Wilkins, 1990; Currie and Tate, 1991) and agent architectures (Firby, 1987; Hunsberger and Zancanaro, 2000).

Our approach to acquiring task models is based on the conjecture that it is more difficult for people to deal with abstractions in the task model than to generate and discuss examples of how to accomplish tasks. In other words, we are developing a *programming by demonstration* system in which a domain expert performs some task by executing primitive actions and then reviews and annotates a log of their actions.

Towards this end, in this paper, we present and analyze techniques for inferring a hierarchical task model from *partially-annotated* examples of task-solving behavior. We do not, however, address important user-interface and mixed-initiative issues that are inherent to developing a full system.

To annotate an example, an expert must *segment* the log to indicate which subsets of actions contribute to the same subtask, and may optionally mark elements in the log to indicate whether two actions could have occurred in another order, whether an action was optional, or whether two parameters of a primitive action must have the same value.

A conceptual contribution of this paper is to decompose model learning into an *alignment* phase, in which the elements of the annotation examples are mapped to concepts in a task model, and an *induction* phase, in which that model is generalized to be consistent with all the input examples.

Our key algorithmic contribution is a technique for inducing the parameters of intermediate goals and ensuring these parameters are bound by the recipes that achieve them. Our experience has been that parameters and bindings are particularly difficult for people to specify, but essential for constructing effective hierarchical task models. We formalize notions of soundness and completeness, and prove that our algorithm is sound and complete.

Finally, we describe experiments that help determine an appropriate division of labor between the human expert and the learning algorithm. There is a tradeoff between how much information the expert provides in each example and how many examples must be provided. For example, the epert can either annotate the actions in an example which are optional, or can provide other examples in which the optional steps do not appear. We have run experiments in two domains to investigate the impact of providing various types of annotations. Our results suggest a design criteria for mixed-initiative task model learning: the human expert should be asked whether equalities between parameter values in the examples are inherent to the domain or are merely coincidental.

2 Terminology and Formalization

Informally, the input to the learning algorithm is a series of demonstrations; each one shows a particular way to perform a task and suggests others. For example, if the sequence

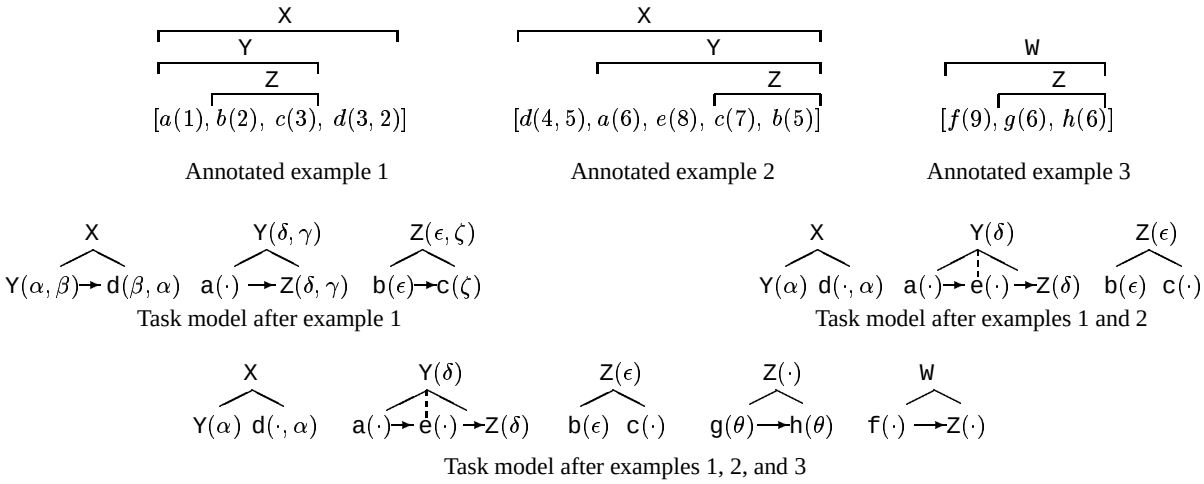

Figure 1: Example of task model learning. The unannotated examples are sequences of instances of primitive actions, shown in lower-case italic letters, such as $[a(1), b(2), c(3), d(4)]$. The action $a(2)$ represents an action of type a with parameter 2. The only annotations in these examples are segmentations 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.

$[a, b, c]$ is correct and b is annotated as optional, then we know $[a, c]$ is also a correct example. We can also generalize from the annotated examples based on assumptions about the target model to be learned. For example, if the learner is told $[a, b, c]$ and $[c, b, a]$ are both correct and the target model represents partial ordering constraints on pairs of actions, all orderings of a, b, and c must be correct. Figure 1 shows an example of task model learning. The model learned after all three examples are processed accepts many action sequences that have not been seen, such as $[a(1), e(2), g(3), h(3), d(1, 2)]$.

2.1 Soundness and completeness

We begin with a general definition of task model learning, in order to formalize the desired properties of our algorithms. Formally, we assume that there is some set $\mathcal{P}$ of primitive actions, a set $\mathcal{E}$ of annotated examples, and a partially-ordered set $\mathcal{M}$ of possible task models. Let $\mathcal{P}^*$ be the set of all finite sequences of the primitives in $\mathcal{P}$. For any annotated example $e \in \mathcal{E}$, let $\hat{e} \in \mathcal{P}^*$ be the unannotated example created by the domain expert. The partial order on $\mathcal{M}$ establishes a preference order on its models. Below, we argue that a preference order is needed for learning minimal parameter information.

A *task model learning algorithm* $\mathcal{A}$ takes a set of annotated examples $\bar{\mathcal{E}} \subset \mathcal{E}$ and returns a model $m = \mathcal{A}(\bar{\mathcal{E}})$. The definitions of soundness and completeness for $\mathcal{A}$ are built upon the notion of a task model being *compatible* with the annotated examples in $\bar{\mathcal{E}}$. The definition of compatibility is specific to the class of models being learned; it is defined for our models of interest at the end of Section 2.3. Given a function compatible(m, e) that returns true or false:

- A task model m is *consistent* with a subset of annotated examples $\bar{\mathcal{E}}$ if consistent$(m, \bar{\mathcal{E}})$ returns true, where consistent$(m, \bar{\mathcal{E}}) \Leftrightarrow (\forall e \in \bar{\mathcal{E}}$, compatible$(m, e))$.
- m is a *preferred consistent model* on $\bar{\mathcal{E}}$ if $\forall m' \in \mathcal{M}$ 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.
- m *accepts* a sequence of primitive actions p^* if accept(m, p^*) returns true, where accept$(m, p^*) \Leftrightarrow \exists e \in \mathcal{E}$ such that consistent(m, e) is true and $\hat{e} = p^*$.
- A task model m is *sound* on $\bar{\mathcal{E}}$ if for all $p^* \in \mathcal{P}^*$, accept$(m, p^*) \Rightarrow (\forall m' \in$ PCM$(\bar{\mathcal{E}})$, accept$(m', p^*))$.
- A task model m is *complete* on $\bar{\mathcal{E}}$ if for all $p^* \in \mathcal{P}^*$, accept$(m, p^*) \Leftarrow (\forall m' \in$ PCM$(\bar{\mathcal{E}})$, accept$(m', p^*))$.

A task model learning algorithm $\mathcal{A}$ is sound and complete if for any $\bar{\mathcal{E}}$, $\mathcal{A}(\bar{\mathcal{E}})$ is sound and complete.

2.2 Task model language

We learn the class of task models used by a middleware system for building collaborative agents (*references omitted for review*). The task model is composed of actions and recipes. Actions are either primitive actions, which can be executed directly, or non-primitive actions (also called "intermediate goals" or "abstract 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 does not have an explicit representation of causal knowledge for its preconditions and effects.

Recipes are methods for decomposing non-primitive actions into subgoals. There may be several different recipes for achieving a single action. Each recipe describes a set of steps that can be performed to achieve a non-primitive action. A recipe also contains constraints that impose temporal partial orderings on its steps, as well as other logical relations among their parameters. For the purposes of this paper, however, we will consider only equality relationships. Equalities between a parameter of a step and a parameter of the action being achieved by the recipe are called bindings, but are otherwise indistinguishable from constraints. All steps are assumed to be required unless they are labelled as optional. Figure 2 contains samples of these representations.

nonprimitive act PreparePasta
  parameter Pasta pasta
recipe PastaRecipe **achieves** PreparePasta
  steps BoilH20 boil
    MakePasta make,
    optional GetItem get
  bindings **achieves**.pasta = make.pasta
  constraints get.item = make.pasta, boil.liquid = make.water
    boil **precedes** make, get **precedes** make
primitive act GetItem
  parameter Item item

Figure 2: Sample representations for a simple cooking domain. Keywords are in bold. Parameters and steps have a name as well as a type in order to allow for unambiguous references in bindings and constraints.

2.3 Annotation language

Annotations allow a domain expert to indicate that examples similar to the one being annotated are also correct examples. An annotated example e is a five-tuple: $\langle \hat{e}, \mathcal{S}, optional, unordered, unequal \rangle$:

$\hat{e}$ is the temporally ordered list of primitive actions $[p_1, \ldots, p_k]$ that constitute the unannotated example demonstrated by the expert.

$\mathcal{S}$ is a segment; a segment is a pair $\langle segmentType, [s_1, \ldots, s_n] \rangle$. Each s_i, called a *segment element* or element for short, is either a primitive action or is a segment. Grouping elements together means that, as a unit, they logically form one occurrence of an intermediate goal of type $segmentType$.

$optional$ is a partial mapping from segment elements to boolean values. If the mapping is defined and is true, the expert is specifying that the example with this element removed would also be correct.

$unordered$ is a partial mapping from pairs of elements in the same segment to boolean values. If the mapping is defined and is true, the expert is specifying that switching the order of appearance of the pair of elements would constitute another correct example from the domain.

$unequal$ is a partial mapping from pairs of action parameters to boolean values. If the mapping is defined and is true, the expert is specifying that another correct example with the same segmentation exists wherein these two parameters do not have the same value.

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. For our task model language, the function compatible(m, e) returns true iff:

- for each segment $\langle type, [s_1, \ldots, s_n] \rangle$ in S, there exists a recipe r in m such that r achieves an action with type $type$ and there exists a one-to-one mapping ϕ from $s_1, \ldots, s_n$ to steps in r such that (1) $\phi(s_i)$ has the same type as s_i, (2) if s_i is mapped to true by $optional$, then $\phi(s_i)$ is marked as optional in r; and is not marked as optional if s_i is mapped to false, (3) if $\langle s_i, s_j \rangle$ is mapped to true by $unordered$ then $\phi(s_i)$ and $\phi(s_j)$ are unordered in r; and are ordered in r if $\langle s_i, s_j \rangle$ is mapped to false, and (4) ϕ maps some s_i to each required step in r,
- for every pair of parameters that $unequal$ indicates can have different values, a plan can be formed from the recipes in m that results in a sequence of primitive actions p^* such that p^* is identical to $\hat{e}$ except for parameters values and the corresponding parameters in p^* have different values.

3 Learning algorithm

Figure 3 contains pseudo code for our task model learning algorithm, which requires polynomial time.

LEARNMODEL $(\bar{\mathcal{E}}) \equiv$
  $m_0 \leftarrow$ ALIGNMENT$(\bar{\mathcal{E}})$
  $m_1 \leftarrow$ 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

A fundamental search problem, which we refer to as alignment, faced by any task learning algorithm is to determine which primitive actions, possibly in different examples, correspond to the same recipe step. Additionally, algorithms for learning hierarchical task models must also match segments to recipes. This is a fundamental problem because a learning algorithm needs to identify segments with recipes and segment elements with recipe steps in order to update the model.

Suppose, for example, that a human expert indicates that $[a, b, c]$ and $[c, b, a]$ both achieve goal Z. The alignment question, here, is whether to learn one or two recipes for Z. Without an assumption or heuristic, there is no justification to learn only one recipe. But if we never combine multiple examples into one recipe, we can not perform any useful generalization.

Alignment is intractable in the absence of assumptions about the domains being studied. However, we render the alignment problem tractable by making the following fairly benign 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.

The ALIGNMENT function constructs a model m with nonprimitive actions without parameters and recipes with only required steps. It also constructs an *alignment* from the annotated examples $\bar{\mathcal{E}}$ to m that consists of a pair of mappings $\langle \sigma, \phi \rangle$ where ϕ maps from each segment in each $e \in \bar{\mathcal{E}}$ to a recipe in m, and σ maps from each element in each segment in each $e \in \bar{\mathcal{E}}$ to a step in a recipe in m. These mappings are used by the induction algorithms.

ALIGNMENT first partitions the segments in $\bar{\mathcal{E}}$ into sets of segments that must, under our assumptions, be mapped to the same recipe. In particular, it groups the segments such that any two segments, s_i and s_j, are grouped together if they have the same $segmentType$ and the set of the types of the elements in s_i that are not marked optional are a subset of the set of the types of the elements in s_j that are not marked optional. For any set of annotated examples, there is only one possible such grouping, which can be easily computed.

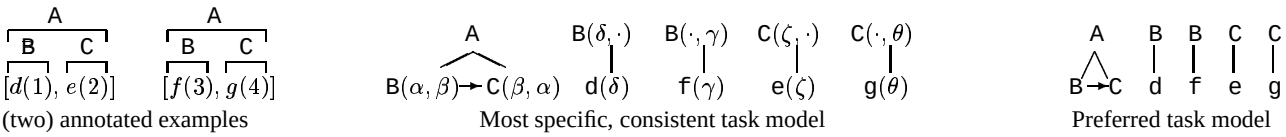

Figure 4: Motivation for suggested parameter bias: the task models differ in that only the first contains propagators that can force d and g's parameters and e and f's parameters to be equal. Both models are consistent because they can each produce both examples. Without the suggested parameter bias,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

Next, for each group of segments, ALIGNMENT creates a recipe with n_t steps of each action type t, where n_t is the largest number of elements of type t that occur in any segment in the group. It maps each segment in the group to this recipe, and maps the segment's elements, in order of occurrence, to steps of the same type in the recipe.

After alignment, our algorithm determines the optionality of, and ordering constraints between, steps. The INDUCEOPTIONAL function marks step s in recipe r as optional if any segment element that is marked optional is mapped to s or if some segment is mapped to r but 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.

We omit the pseudo code of the ALIGNMENT, INDUCEOPTIONAL, and INDUCEORDERING functions since they are not the focus of this paper.

3.1 Inducing propagators

We now present and discuss methods for inferring bindings, constraints, and parameters of non-primitive actions, which we will refer to collectively as *propagators*.

The role of propagators is to enforce equality relationships among the parameter values of primitive actions. For example, in a task model for cooking spaghetti marinara, the cooked pasta must be the same pasta to which sauce is later added. In contrast, different knives can be used to cut, say, the tomatoes and the mushrooms. These equality relations cross the boundaries of many actions and recipes, i.e. they are not local to any particular recipe.

The first step of learning propagators is to decide which parameters values should be forced to be equal. For example, if the same knife is used to cut vegetables in all examples, then we can conclude that the same knife must be used. If we then saw an example in which different knives were used, we would retract this constraint. Alternatively, the annotations can indicate that different knives could have been used.

A problem arises due to pairs of steps that never occur in the same example. As shown in Figure 4, there exist consistent task models that constrain the parameters of these steps to be equal. Such models are counter-intuitive because they postulate elaborate constraints that are not positively suggested by any example. Further, to remove all unnecessary constraints, the learning algorithm must see examples that contain all pairs of steps that are *unrelated* to each other.

To address this problem, we propose a bias against models with unsuggested propagators. A model m's propagator is *suggested* by an example $e \in \bar{\mathcal{E}}$ if: (1) e contains two parameters of primitive actions that are not annotated as unequal and have the same value; (2) there exists another example e' such (a) e' is identical to e except that the two parameters have different values; (b) m is not compatible with e'; and (c) removing the propagator from m produces a model which is compatible with e'. For example, in Figure 4, none of the propagators are suggested because no parameter values are equal in the two examples being modeled. However, in Figure 1, all of the propagators are suggested.

We propose the following bias:

Suggested parameter 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 suggested by an example in $\bar{\mathcal{E}}$ and m_j has propagators that are not suggested by any example in $\bar{\mathcal{E}}$.

Model learning thus benefits from Occam's Razor: the simplest model that explains the data should be preferred. For propagators, we claim the simplest model contains only what is needed to explain the equalities evident in the examples.

Note that an unpreferred 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 suggested and so it would be preferred.

Figure 5 shows pseudo code for an algorithm for learning propagators with two modes, one that is sound with no preference bias and one that is sound under the suggested parameter bias. The algorithm takes as input the annotated examples and a task model that lacks some or all of these elements, and produces a more complete task model.

A data structure that is used to facilitate 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 that has a non-empty sequence of steps, STEP(p) returns the last recipe step in the sequence, TAIL(p) returns a path identical to p except that STEP(p) is absent, 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 bindings 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.

```
INDUCEPROPAGATORS (m,ℰ̄) ≡
  forall R in ALLRECIPES(m)
    ADDCONSTRAINTS(R,ℰ̄)
ADDCONSTRAINTS (R,ℰ̄) ≡
  ℒ ← ∅
  𝒫 ← NONRECURSIVEPATHSTORECIPE(ℛ)
  forall p in 𝒫
    ℒ ← ℒ ∪ PATHPAIRINGS(p, 𝒫,ℰ̄)
  forall L in ℒ
    forall pairs p, p′ in L
      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, inName) ≡
  tail ←TAIL(p)
  if tail has no steps
    then pName ← NAME(START(p))
    else
      pName ← GENSYM()
      PROPAGATENAME(tail, pName)
  if inName ≠ null
    ℛ ← RECIPE(p)
    add a parameter named inName of type TYPE(START(p))
      to PURPOSE(ℛ)
    add a binding between parameter inName of PURPOSE(ℛ)
      and parameter pName of STEP(p) to ℛ
  return pName
PATHPAIRINGS (p, 𝒫,ℰ̄) ≡
  ℒ ← ∅
  forall p′ in 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 Suggested Parameter Bias is not in effect
      then ℒ ← ℒ ∪ {{p, p′}}
  return ℒ
```

Figure 5: Pseudo code to infer propagators

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 this example implies that a correct 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 Suggested Parameter 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 unsuggested 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 much information the expert provides in each example and how many examples must be provided. 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 propagators. Additionally, in lieu of our restrictions of the model language, annotations can directly provide recipe and step names. Also, while the INDUCEPROPAGATOR algorithm we presented produces an inordinate number of propagators, our implementation reuses propagators when possible. Our deployed system will, of course, have to allow task models to be edited in order to give semantically meaningful names to recipes and steps.

For our experiments, we start with a target task model and use it as an oracle to both generate and annotate test examples. After each example, we determine if the algorithm has produced a task model equivalent to the target task model given the examples it has seen. Additionally, after each example, we compute the *error rate*, i.e., a measure of how different the current task model is from the target model. Finally, we determine if each example was *useful*, i.e. if it contained any new information that was not implied the previous example, by seeing if the algorithm's internal data structures were altered.

We ran our experiments on two target 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 an artificial 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 are always provided by the oracle, but we varied whether the other annotations were provided. We ran all variations of possible combinations of annotation types, and report a subset in Table 1. In the table, O indicates that all ordering annotations are given, E indicates that all equality annotations are given, and P indicates that all propagators are given. Annotating optional

Annotation	Cooking			Symbol Editor		
	Avg.	Min.	Useless	Avg.	Min.	Useless
All	5.3	3	9.9	1.9	1	0.1
EOP	6.5	3	11.1	2.4	1	0.4
EP	7.2	4	14.1	3.0	2	0.5
EO	7.2	3	10.4	14.2	3	47.0
E	8.1	4	13.1	14.4	3	46.9
O	38.3	15	404.3	53.0	37	118.7
None	38.3	15	404.2	53.1	37	118.6

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

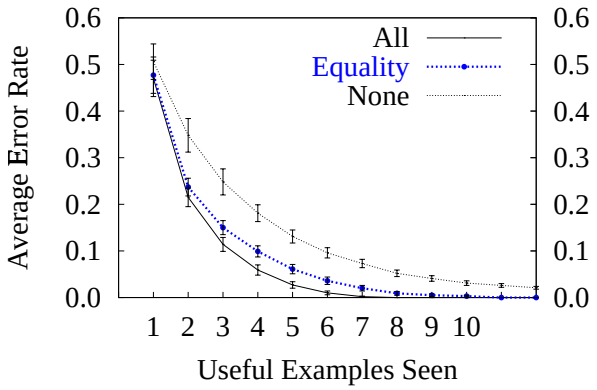

Figure 6: Algorithms quickly produce a nearly correct model

steps did not significantly impact the results, except when all other annotations were given (indicated by 'All' in the table). The reason for this is that optionality is the easiest aspect to learn because it does involve relationships between steps. Note that annotating equalities does not add any information when propagators are given. The reported values are averaged over randomized sequences of examples — 100 trials for the cooking domain and 20 trials for the Symbol Editor.

The main surprise is that providing equality annotations dramatically reduces the number of required examples. This is interesting because it seems likely that it will be much less onerous for a human expert to indicate when apparent equalities in the example are coincidental, than to construct all the propagator information directly.

Another interesting result in Table 1 is 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 trial (the "min" column) and the average number of useless examples per trial (the "useless" column). One could imagine that a human expert would provide examples closer to the minimum than to the average and would not present useless examples.

Figure 6 shows the error rate as a function of the number of useful examples seen. The error rate is measured as the fraction of the total information that remains to be learned. The graph 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

Bauer (Bauer, 1998, 1999) presents techniques for acquiring non-hierarchical task models from unannotated examples for the purpose of plan recognition (i.e., inferring a person's intentions from her actions). 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.* (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 agent 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 and bindings for intermediate goals, 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 the intelligent tutoring systems, through both demonstration and automated experimentation in a simulated environment. Masui and Nakayama (1994) investigates learning macros from observation of or interaction with a computer user in order to assist the user with tasks that occur frequently or are inherently repetitive. Lau *et al.* (2000), in one of the few formal approaches to learning macros, uses 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.

References

Richard Angros Jr. Agents that learn what to instruct: Increasing the utility of demonstrations by actively trying to understand them. Technical report, Univ. of Southern California, 2000.

Mathias Bauer. Acquisition of Abstract Plan Descriptions for Plan Recognition. In *AAAI98*, pages 936–941, 1998.

Mathias Bauer. From Interaction Data to Plan Libraries: A Clustering Approach. In *Proc. of the 16th Intl. Joint Conf. on AI*, 1999.

K. Currie and A. Tate. O-plan: the open planning architecture. *Artificial Intelligence*, 52(1):49–86, 1991.

R. J. Firby. An investigation into reactive planning in complex domains. In *Proc. 4th Nat. Conf. AI*, pages 202–206, 1987.

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

Luke Hunsberger and Massismo Zancanaro. A mechansim for group decision making in collaborative activity. In *Proc. 17th Nat. Conf. AI*, pages 30–35, 2000.

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 S. Weld. Version space algebra and its application to programming by demonstration. In *Intn. Conf. on Machine Learning*, pages 527–534, 2000.

T. Masui and K. Nakayama. Repeat and predict– two keys to efficient text editing. In *Conference on Human Factors in Computing Systems*, pages 118–123, 1994.

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

D. Wilkins. Can AI planners solve practical problems? *Computational Intelligence*, 6(4):232–246, 1990.