A New Probabilistic Plan Recognition Algorithm Based on String Rewriting

Christopher W. Geib
School of Informatics
University of Edinburgh
10 Crichton Street,
Edinburgh, EH8 9AB, Scotland
cgeib@inf.ed.ac.uk

John Maraist and **Robert P. Goldman**
SIFT, LLC
211 North First Street, Suite 300
Minneapolis, MN 55401-1480
{jmaraist,rpgoldman}@sift.info

Presented at the Eighteenth International Conference on Automated Planning and Scheduling (ICAPS-2008), Sydney, Australia, September 14–18, 2008.

Abstract

This document formalizes and discusses the implementation of a new, more efficient probabilistic plan recognition algorithm called Yet Another Probabilistic Plan Recognizer, (Yappr). Yappr is based on weighted model counting, building its models using string rewriting rather than tree adjunction or other tree building methods used in previous work. Since model construction is often the most computationally expensive part of such algorithms, this results in significant reductions in the algorithm's runtime.

Introduction

Given a library of plans, *plan* or *intent recognition* is the process of identifying which of the plans an agent is performing based on a sequence of observations of their actions. Much of the prior AI research in plan recognition has been based on hierarchical models of plans (Geib & Goldman 2003; Avrahami-Zilberbrand & Kaminka 2005; Bui, Venkatesh, & West 2000; Kautz 1991). These systems use tree based data structures to represent plans, and often use tree traversal (Avrahami-Zilberbrand & Kaminka 2005; Kautz 1991) or tree adjunction (Geib & Goldman 2003) algorithms to build and maintain plan hypotheses.

The PHATT system (Geib & Goldman 2003) (hereinafter G&G) performed plan recognition by weighted model counting, using tree adjunction of left-most, depth-first, plan derivation tree fragments to construct models that explained the observed actions. In this paper, we argue tree adjunction is inefficient. Instead, we formalize and argue for a new probabilistic algorithm for plan recognition based on model counting called Yet Another Probabilistic Plan Recognizer (Yappr). Yappr differs from PHATT in building more compact explanatory models based on a form of string rewriting maintaining only the plan tree frontiers.

Problem Intuition

All probabilistic plan recognition systems based on weighted model counting function in roughly the same way. These systems compute the exclusive and exhaustive set of models that can explain a given set of observations. They then assign a probability to each such model and then compute the likelihood of a particular goal by summing the probability of those explanatory models in which it occurs. Since models in these systems explain the observations in terms of plans that are being executed, we will use: *model*, *explanation*, *explanatory model*, and *hypothesis* interchangeably.

PHATT's models contain collections of partially expanded hierarchical plans(Ghallab, Nau, & Traverso 2004) for specific goals that account for the observed actions. PHATT builds these plans incrementally, from left to right, by adjoining left-most, depth-first plan tree fragments to the existing plans. To extend an existing plan these left-most trees are attached at the plan's *frontier*. Keep in mind that there may be multiple ways to incorporate a single observation into an explanatory model. The different ways of incorporating the observation yield multiple trees that could be adjoined into the model. In such cases, PHATT makes multiple copies of the original model, and each tree is inserted into its own copy of the plan structure expanding the set of explanations.

Note that during explanation construction, the part of the plan's tree structure above the frontier is not used, but copying that structure imposes significant computational costs. It is the frontier (in combination with the observation) that defines which new trees can be added to the model. Further, all additions to the explanation's structure are made at the frontier. We will see that with simple bookkeeping we do not even need the tree structures to extract the plan's root level goals. Since all interactions with the explanation take place at the frontier, we can dispense with the tree structure, and maintain only the tree frontier. This avoids creating and copying unnecessary interior tree structure in the model's plans.

We will show that the frontier of the trees in an explanatory model can be represented as a string of non-terminals and ordering constraints from a grammar that defines the original plan library to be recognized. By careful precomputation, we can build a plan grammar that explicitly models only the frontiers and ordering constraints of the left-most depth-first trees from the original plan library. We will then show how to use the new grammar to build explanatory models that maintain just the frontiers of the plans in the model. This will allow us to build and maintain equivalent but significantly smaller model structures.

The plan recognition community distinguishes between *plan* recognition, the recognition of the complete plan structure being followed by the agent, and *goal* recognition, where only the top-level goals of the agent are identified. Blaylock (2005) provides a review of these competing approaches. We choose to take a middle road. Since the root goals being pursued are the most common probabilistic query for such systems, we argue for an algorithm that builds restricted models that allow us to quickly answer this question while still allowing the system to reconstruct complete plan models if a probabilistic query is made about details not available in our restricted models.

Plan Recognition as Weighted Model Counting

To more clearly see plan recognition as weighted model counting, we can informally define an explanation, e, for a set of observations, obs, as a set of plans, $p_0, ...p_i$ each built to achieve a goal, $g_0, ...g_i$ from a set of known goals, G . There may be multiple such explanations for fixed obs that differ in the goals, plans being pursued, or assignment of observations to plans. Leaving the observations implicit, we denote the complete and covering set of such explanations for a given set of observations as Exp and the subset of Exp that make use of a particular goal $g \in G$ as Exp_g

Since we are taking a Bayesian approach to plan recognition, we want to compute the conditional probability of a particular goal given the set of observations $Pr(g|obs)$. Using Bayes' rule:

$$Pr(g|obs) = Pr(g \wedge obs)/Pr(obs)$$

This can be rewritten as the equivalent formulation:

$$Pr(g|obs) = Pr(g \wedge obs)/\sum_{\forall g \in G} Pr(g \wedge obs)$$

where the denominator is the sum of the probability mass for all goals. However given our definition, we know that the plans in an explanation determine the goals in that explanation. Given that Exp is complete and covering, we can rewrite the previous equation as:

Formula 1

$$Pr(g|Obs) = \sum_{e \in Exp_g} Pr(e \wedge Obs)/\sum_{e \in Exp} Pr(e \wedge Obs)$$

where the denominator sums the probability of all explanations for the observations, and the numerator sums the probability of the explanations in which the goal g occurs.

Thus, if we can compute the probability of individual explanations for the observations, we can perform plan recognition by weighted model counting. We build a mutually exclusive and exhaustive set of possible explanations for the observations, compute the probability for each, and then sum the probability mass of explanations that contain a particular goal and divide by the probability mass of all of the explanations of the observations.

Algorithm Intuitions

Intuitively, the approach we will argue for with Yappr has three high level steps:

1. Offline precompile the plan library into a **plan frontier fragment grammar** (PFFG).
2. Use the resulting PFFG to build explanations for the observed actions. This process will look very much like a string rewriting system or a graph editing process.
3. Maintain the PFFG productions and the order in which they are used in building each of the explanations. This information will allow us to build more complete explanations in the case of queries for information beyond the root goals being followed.

This approach to plan recognition is similar to parsing methods that directly build a logical form rather than building a parse tree (Steedman 2000). However, traditional parsing methods cannot be used for this task. The most pressing reason for this is our commitment to allowing for the execution and recognition of multiple, concurrent, interleaved plans. Consider the case of a person taking care of a small child, doing laundry, and cleaning the kitchen all at the same time. People regularly engage in such multitasking; our plan recognition algorithms should be capable of recognizing these situations and their component plans.

However, standard parsing algorithms like CKY (Younger 1967) are designed for a single sentence at a time and usually assume that they will observe a complete sentence. Human language speakers do not interleave multiple sentences in the same way they interleave execution of plans. It is therefore not surprising that existing parsing algorithms make these assumptions. However, reliance on these assumptions does make them inappropriate for our task.

In the rest of this document we will: define PFFGs; fully specify Yappr's algorithm for building the explanations on the basis of PFFGs; describe how the probabilities for each explanation can be computed over explanation structures; provide a complexity analysis; and finally compare Yappr empirically to PHATT.

Building PFFGs

The foundation of any plan recognition system is a collection of plans to be recognized. These plans must be specified in a formal language.

Definition 1 *We define a* plan library *as a tuple* $PL = \langle \Sigma, NT, R, P \rangle$ *where,* Σ *is a finite set of basic actions or terminal symbols,* NT *is a finite set of methods or non-terminal symbols,* R *is a distinguished subset of "intendable root" non-terminal symbols* $R \subseteq NT$*, and* P *is a set of production rules of the form* $A \rightarrow \alpha : \phi$ *where:*

1. $A \in NT$
2. α *is a string of symbols from* $(\Sigma \cup NT)^*$
3. $\phi = \{(i, j) | \alpha[i] \prec \alpha[j]\}$ *where* $\alpha[i]$ *and* $\alpha[j]$ *refer to the* i^{th} *and* j^{th} *symbols in* α*, respectively.*

Intuitively this definition is a traditional context free grammar(CFG) with the addition of explicit ordering constraints, defined by a relation, $\prec$. Ordering constraints are added to each of the productions of the grammar to indicate when actions must be performed in a specific order. Traditional CFGs assume that ordering of symbols indicates a required

ordering in the plan. Here we assume that all actions on the right hand side of a production rule are unordered unless the ordering relation states otherwise.

In a traditional CFG partial ordering is handled by replicating those rules that contain partial ordering, one production rule for each of the possible orderings for the actions. Since a system of the kind we are describing would be forced to consider all of the possible orderings for the actions, using a traditional CFG could result in a catastrophic increase in the computational cost of the algorithm.

This formulation is also subtly different than a traditional Hierarchical Task Network (HTN) plans (Ghallab, Nau, & Traverso 2004). Some formulations of HTNs allow for arbitrary applicability conditions that must be true before a production can be used. The plan library defined here is strictly less expressive than these formulations of HTNs, but equivalent to HTN formulations without these additional conditions.

Definition 2 *Given a rule $\rho = A \rightarrow \beta : \phi$ We say $\beta[i]$ is a **leftmost child of** A **given** ρ if $\nexists j$ such that $(j, i) \in \phi$.*

The set of leftmost symbols for a given rule allows us to talk about those symbols that are required to be first in any expansion of the parent non-terminal by that rule. Note that this definition does not require that there be a unique leftmost symbol of a rule, and we denote the set of such symbols as $\mathcal{L}(\rho)$. We will use $\mathcal{R}(\rho)$ to denote the set of all symbols that are not leftmost of ρ.

Definition 3 *Given a plan library $PL = \langle \Sigma, NT, R, P \rangle$, we define a **leftmost tree T, deriving** α, as a tree such that,*

1. *every node in T is labeled with a symbol from $\Sigma \cup NT$.*
2. *every interior node in T is labeled with a symbol from NT.*
3. *if an interior node, n, labeled A in T has children with labels $\beta_1, ...\beta_k$, then*
   - *$\exists \rho \in P | \rho = A \rightarrow \beta_1 ...\beta_k : \phi$,*
   - *node n is additionally annotated with ρ*
   - *no children of n labeled with symbols in $\mathcal{R}(\rho)$ have children of their own.*
   - *at most one child of n labeled with a symbol in $\mathcal{L}(\rho)$ have children of its own.*
4. *there is a distinguished node in the frontier of T labeled with the terminal symbol α that is leftmost for its parent. We call this the **foot** of the tree T.*

Leftmost trees correspond very closely to minimal, leftmost, depth-first, derivation trees for a specific terminal in traditional CFGs. In this case, the ordering relation defined for the plan library is used to determine which non-terminals are leftmost. We will use leftmost trees to build PFFGs. To do this, we first define a *generating* set of trees:

Definition 4 *A set of leftmost trees is said to be **generating** for a plan library $PL = \langle \Sigma, NT, R, P \rangle$ if it contains all of the leftmost trees that derive some basic action in Σ rooted at a method in NT. We denote the generating set $\mathcal{G}(PL)$ and refer to its members as **generating trees**.*

Finally, on the basis of these generating trees we can define the PFFG for the specific plan library.

Definition 5 *We define the **Plan Frontier Fragment Grammar(PFFG)** for the plan library $PL = \langle \Sigma, NT, R, P \rangle$ and its generating trees $\mathcal{G}(PL)$ as a tuple $PFFG_{PL} = \langle \Sigma, NT', R, P' \rangle$ where $NT' \subseteq NT$ and P' is the set of all production rules of the form: $pid : \langle a, B \rangle \rightarrow \alpha : \phi$ for a tree $T \in \mathcal{G}(PL)$:*

1. *pid is a unique identifier for the production rule,*
2. *$a \in \Sigma$ and a is the foot of T,*
3. *$B \in NT'$ and B is the root of T,*
4. *α is the string of symbols from $(\Sigma \cup NT)^*$ that are the frontier of T with a removed.*
5. *$\phi = \{(i, j) | \alpha[i] \prec \alpha[j]\}$ where $\alpha[i]$ and $\alpha[j]$ refer to the i^{th} and j^{th} symbols in α respectively and T requires that $\alpha[i] \prec \alpha[j]$.*

Effectively, the preceding set of definitions has used a particular plan library to produce the set of leftmost trees for the plans and then defined a grammar that captures just the frontiers of the leftmost trees used in the construction of any of the original plans. In so doing, we have precompiled information about the structure and choices inherent in the plan without requiring the grammar be totally ordered. This way of thinking about the plan library will allow us to maintain the state of the derivation of a particular explanation by just maintaining the frontier of the explanation along with its ordering constraints.

For these definitions to result in a finite PFFG we bound any recursion in productions of the plan library. If this is not done, the number of generating trees is unbounded, resulting in an infinite PFFG. To ease our analysis of the complexity of the algorithm, we will also assume, without loss of generality, that the initial plan library's production rules contains no epsilon productions (Aho & Ullman 1992). This does not present a significant issue as they can be removed before production of the PFFG using the traditional method for CFGs. Since the process of producing the PFFG cannot introduce epsilon productions, the resulting PFFG will not have them either.

Building Explanations

Having constructed a PFFG for a particular plan library we now specify how we use this to build explanations.

Definition 6 *We define a (possibly partial) **explanation** for a sequence of observations $\sigma_1 ...\sigma_n$, given a plan library $\langle \Sigma, NT, R, P \rangle$, as a tuple $\langle \sigma_i ...\sigma_n, \alpha, \phi, PS \rangle$ where $\sigma_i ...\sigma_n, i \geq 1$ are those observations that have not yet been explained, $\alpha \in (\Sigma \cup NT)^*$ represent the frontiers of the plans in the explanation (we will call this the **explanation frontier**), $\phi = \{(j, k) | \alpha[j] \prec \alpha[k]\}$ where $\alpha[j]$ and $\alpha[k]$ refer to the ordering constraints on the j^{th} and k^{th} symbols in α respectively, and PS is a sequence of pairs of production identifiers and elements of α, $\langle pid_1, \alpha[x] \rangle, \langle pid_2, \alpha[y] \rangle, ..., \langle pid_p, \alpha[z] \rangle$ that record the specific productions that were applied to produce α.*

Next we define an explanation's *pending attachment points*.

Definition 7 *We define the **pending attachment points** for an explanation $e = \langle \sigma_i ...\sigma_n, \alpha, \phi, PS \rangle$ as $\{\alpha_k | \alpha_k \in \alpha \wedge \nexists (j, k) \in \phi\}$ and we denote this set as $AP(e)$*

Using this set of definitions we can now describe the algorithm for building the complete set of explanations for a given set of observations. We define the initial explanation for every problem as the tuple $\langle \sigma_1 ... \sigma_n, \{\}, \{\}, \{\} \rangle$, where $\sigma_1 ... \sigma_n$ is the set of all the observations to be explained, and the other elements of the explanation are empty. Given this base case, we process each observation creating new explanations as necessary.

There are three possibilities for each observation: 1) the observation removes a single terminal symbol from the explanation, 2) the observation adds non-terminals to the frontier of the explanation, and 3) the observation is the first action of a previously unobserved plan. The following pseudo code produces the complete set of explanations for a given a set of observations and a PFFG. Each case is commented in the code.

```
PROCEDURE Explain({σ1...σn, PFFG_PL = ⟨Σ, NT, R, P⟩})
  D0 = {}; E = Emptyqueue();
  Enqueue(⟨σ1...σn, {}, {}, {}⟩, E);
  %% For each observation
  FOR i = 1 to n DO
      %% Loop over all the explanations.
      WHILE Nonempty(E) DO
          E′ = Emptyqueue()
          e = ⟨σi...σn, α, φ, PS⟩ = Dequeue(E);
          %% Extend existing plans.
          FOR EACH B ∈ AP(e);
              %% removing terminals from the frontier
              IF B = σi, THEN
                  α′ = α − B;
                  φ′ =UpdateRemoving(φ, B);
                  Enqueue(⟨σi−1...σn, α′, φ′, PS⟩, E′);
              END IF;
              %% expanding the frontier
              FOR EACH pid : ⟨σi, B⟩ → γ : ψ ∈ P
                  α′ = (α − B) + γ;
                  φ′ =UpdateAdding(φ, ψ);
                  PS′ = PS + ⟨pid, B⟩;
                  Enqueue(⟨σi−1...σn, α′, φ′, PS′⟩, E′);
              END FOR EACH LOOP;
          END FOR EACH LOOP;
          %% Introduce new plans.
          FOR EACH pid : ⟨σi, C ∈ R⟩ → γ : ψ ∈ P
              α′ = α + γ;
              φ′ =UpdateAdding(φ, ψ);
              PS′ = PS + ⟨pid, C⟩;
              Enqueue(⟨σi−1...σn, α′, φ′, PS′⟩, E′);
          END FOR EACH LOOP;
      END WHILE LOOP;
      E = E′
  END FOR LOOP;
RETURN E;
```

A small amount of bookkeeping must be done in each case to make sure the ordering constraints are kept consistent and those that refer to a removed actions are deleted or appropriately redirected to existing actions in the explanation. This is the task of the *UpdateRemoving* and *UpdateAdding* functions. Note that if none of these cases apply, the current explanation is inconsistent with the current observation and is pruned from the search.

Note the addition of $\langle pid, B \rangle$ to PS in the second and third cases. If required, we can use PS, with the ordered list of observations, to recreate the entire tree structure. Walking the list of pid, non-terminal pairs and adjoining a copy of the tree structures from the plan library indicated by the *pid* at the specified non-terminal, will allow us to reconstruct the full plan structure underlying the explanatory model.

It is the fact that B is *removed* or *replaced* in α by γ, or that γ is added to α that makes this algorithm very much in the spirit of a string rewriting algorithm. The string of symbols, α, representing the explanation frontier are rewritten by the PFFG rules.

Keep in mind that each enqueue operation in the above code creates a duplicate of the explanation structure to allow the different rules to be applied separately. It is this copying of the explanation structures that is less time consuming with Yappr's structures than with the forest of trees that PHATT uses. We will return to discuss this in detail later.

This completes the discussion of the definitions for PFFGs and an algorithm for building explanatory models based on them. To complete our algorithm, we must define a way of computing the probabilities of these models.

Computing Probabilities for an Explanation

Recall from our earlier discussion of Formula 1 that to compute the conditional probability for any particular root goal, we need to be able to compute the probability of each explanation and the observations. To do this, we use the following formula, which is very similar to the one used in G&G.

Definition 8

$$Pr(exp \wedge obs) = \prod_{i=0}^{I} Pr(G_i) \prod_{i=1}^{n} (Pr(rule_i) \prod_{k=1}^{n} (1/|ext(AP(exp_k)|)$$

where there are n observations. We discuss each term below.

The system has to be provided with a prior probability for each of the root goals in the explanation. The product of these priors is the first term of the above formula. It is straightforward to compute this given an explanation's PS. From the set of productions in PS, we can extract the root goals in the explanation. The system then takes the product of these goal's priors.

For each PFFG rule, we must precompute the probabilistic contribution of any plan choices that are captured within the rule. Remember that any particular PFFG rule may actually encode the choice of multiple productions from the original plan library. The likelihood of the agent making each of these choices must be captured in the model's probability. Modeling the agent's decision making process for this is a very complex problem, and nothing in our approach rules out complex probability models for this. However, for simplicity and low computational cost in our implementation, we have assumed that each of these choices is captured by a uniform distribution. The probability for each PFFG

rule is computed off line when the PFFG is created and associated with its respective rule. This allows the second term in the above equation to be computed by taking the product of the probabilities associated with each production used in the explanation.

The final term captures the likelihood that a particular action is chosen next from the set of all actions consistent with the plan library, the agent's goals, and the actions they have already executed. This is equivalent to choosing one rule from the set of all the production rules that could be applied given the pending attachment points. We denote the set of such rules for a given explanation, exp, at a given time point, t, as $ext(AP(exp_t))$. This is just the set of production rules in the PFFG with left hand sides $pid : \langle \sigma', B \rangle$, where σ' matches the next observation and $B \in AP(exp_t) \cup R$.

Again, while a very complex model of the agent's decision making process could be used to select between these rules, we have assumed that all of the production rules that could be performed at any point are equally likely. This reduces the final term to knowing the number of such rules, and we denote this by $|ext(AP(exp_t))|$. This means the probability of any given rule being selected is just one over this number. To derive $|ext(AP(exp_t))|$ we can simply count the number of explanations enqueued for each trip through the while loop. This counter and associated book keeping are not included in the pseudo code for clarity of presentation.

Note there is a subtle point about this case. When a new goal is introduced it is necessary to modify this number for each preceding time point. Since our model is based on the assumption that the set of goals is determined before the actions are started, when a new goal is first observed it is possible that any of the lead actions for the new plan could have been done earlier. This means that when we add a new goal to an explanation we need to account for the possibility that it could have been performed earlier, and this requires modifying the recorded sizes of the pending attachment point sets. Note however, this is a $O(n)$ operation where n is the number of observed actions, and as we will see, it is dominated by the cost of explanation construction.

On the basis of these three pieces of information Yappr computes the probability of each explanation and the observation. Then, using Formula 1, Yappr is able to compute the conditional probability of any goal.

Complexity Analysis

Having given the Yappr algorithm, we still need to show that it will be more efficient than model construction based on tree adjunction. We will show this by considering the complexity of the model construction algorithm.

In Yappr, the complexity for generating a single explanation is $O(nlog(m))$ where n is the number of observations to be explained, and m is the length of the longest plan possible in the plan library. We argue for this in the following way. For a single explanation, for each of the n observations, a single PFFG rule is instantiated, the non-terminal is removed from the explanation, and the right hand side of the instantiated PFFG rules is inserted.

With efficient data structures the removal of the non-terminal and the insertion of the right hands side of the PFFG rule can be done in constant time, however the instantiating of the PFFG rule requires creating a copy of the rule. This process is dominated by the copying of the right hand side of a PFFG rules. The length of the right hand side of the any PFFG rule, corresponds to the depth of the original plan tree, and so costs $O(log(m))$ to copy and instantiate.

Note the $O(log(m))$ length of the rules holds even if there are multiple non-terminals at each level of the leftmost trees that generated the PFFG. To see this, let K to be the maximum length of any of the production rules in the initial library. This means that any individual level of one of the leftmost trees can have no more than K non-terminals and by extension the length of any rule in the PFFG can be no longer than $Klog(m)$. $K \leq m$ and for most domains $K << m$. This only expands the PFFG right hand by a constant and the depth of the original plan tree dominates this feature for all domains where $K << m$.

Given that the PHATT algorithm is adjoining leftmost trees, its complexity is not significantly different for this portion of the problem. To see this, consider that the significant difference between a single leftmost trees and a PFFG production rule is the addition of a $O(log(m)$ number of non-terminals that act as a spine for the attachment of the frontier symbols. We do note that the constant for the Yappr algorithm should be significantly smaller.

As we argued in the introduction, the significant savings for Yappr is in the smaller size of the data structure it uses to represent the plans in an explanation. In Yappr, as in PHATT, the creation of each explanation requires copying the explanation structure. Since in the worst case there can be an exponential number of explanations to consider (Geib 2004) the size of the data structures to be copied is critical.

In PHATT each explanation is a forest of tree structures, in the worst case a single tree of $O(2^n - 1)$ nodes has to be copied for each of the explanations. Bounding the size of the Yappr data structure is a little more challenging. In order to copy an explanation in Yappr, we must copy both the model's frontier and the ordering constraints.

First, we consider the ordering constraints. If we let the size of the explanation frontier be m, then in the worst case there is an ordering constraint between each pair of actions in m, resulting in m^2 constraints that need to be copied. Given the absence of epsilon productions in the initial grammar, the size of the explanation frontier must be less than or equal to the number of observations. Therefore in the worst case, copying the ordering constraints would take $O(n^2)$.

Note that given the absence of epsilon productions the frontier itself can never exceed n elements in length making the copying of the frontier $O(n)$, and dominated by the constraint copying process. Therefore the worst case $O(n^2)$ sized copy operations for our algorithm represents a significant savings over the tree copying algorithm.

For less densely connected graphs the effective complexity of the Yappr data structure will be less that $O(n^2)$ and the resulting savings will be greater. Most promising for the Yappr algorithm, the greatest number of explanation copy operations occurs precisely when the actions in the plan are least ordered. Plans with completely unordered actions result in a very large number of possible explanations. How-

ever, in precisely these cases, the Yappr explanation copying operation reduces to $O(n)$ since only the frontier needs to be copied and there are no ordering constraints. Thus the greatest computational savings from the smaller size of the copy operations for Yappr occurs exactly in the most computationally expensive cases. Our experiments confirm this.

We also consider the cost of computing the probability of an explanation in the Yappr data structure. Inspection of Definition 8 shows that we have to perform operations on each of the rules used. These multiplications can be done in a single pass as can the computing the probability of the root goals. Thus, the cost of computing the probability for a single explanation in Yappr is $O(n)$, and is dominated by the construction of the explanations.

Experimental results

To strengthen and verify the complexity results of the previous section, we implemented Yappr in Allegro Common Lisp (ACL) and ran empirical studies to directly compare the runtime of the PHATT and Yappr systems on the same input data. Our experiments were conducted on a single machine, a dual AMD 2000+ (1666.780) running Kubuntu Linux 7.04. We used ACL's timing facilities to measure the CPU time of the algorithms alone, excluding garbage collection and system background process activity.

All of our experiments began by generating a plan library to be recognized. These libraries were represented as partially ordered and/or trees that are similar to HTNs (Ghallab, Nau, & Traverso 2004). In this case and-nodes in the tree correspond to HTN methods and or-nodes correspond to the presence of multiple methods for a single goal in the plan. For all plans the root node was defined to be an or-node, and the plan alternated layers of or-nodes and and-nodes.

There are a number of features that define a plan library:

- *Root goals*: The number of top-level root goals.
- *Plan depth*. The depth of each plan tree in the library. Our plan libraries are made up of alternating layers of and-nodes and or-nodes, we define a depth of one to mean each plan has one layer of or-nodes followed by one layer of and-nodes. A depth of two means there are two such two-ply layers.
- *And-node branching factor.* The number of children for each and-node. Note this factor when combined with the plan depth determines the length of each plan in the library. For example, a plan with depth two and and-node branching factor of three has nine (3^2) observable actions.
- *Or-node branching factor.* The number of children for each or-node. Since or-nodes represent alternatives in the plan, this factor has no effect on the length of plan instances but along with the plan depth and and-node branching factor determines the number of possible plans for each root goal in the library.
- *Order.* Within each and-node the order factor determines if and how the siblings are related via causal ordering links. We will examine four possible ordering conditions.
  - *Total*. Children of an and-node are totally ordered. Each child action has a single ordering constraint with the child action that precedes it.
  - *First*. The children of each and-node have a designated first action. All other sibling actions are ordered after it but are unordered with respect to each other.
  - *Last*. The children of each and-node have a designated last action. All other sibling actions are ordered before it, but are unordered with respect to each other.
  - *Unord*. The children of each and-node are completely unordered with respect to each other.

It is this last feature, order, that we treat as an experimental factor here. We hold the rest of these features constant with the following values:

Root goals	100
Plan depth	2
And-node branching factor	3
Or-node branching factor	2

For each of the different values of the order factor, we generated 100 data points by randomly selecting one root goal from the plan library and generating a plan for that root goal that obeyed the rules and ordering constraints in the plan library. Thus, each data point was nine observations long and contained no noise. Each such list of observations was then presented in turn to both Yappr and PHATT to compute the conditional probabilities for the goals. To provide the most equivalent runtime environments possible for each invocation of each algorithm, we triggered a full garbage collection beforehand. For each run we imposed a thirty second timeout on the problem. The runtimes for each test case are presented in Figure 1, and summarized in Table 1.

Order	Algorithm	Mean	Std. dev.
Unord	Yappr	15728	78.4
First	Yappr	6.6	2.31
	PHATT	19.3	2.92
Last	Yappr	20.3	7.54
	PHATT	192.7	49.1
Total	Yappr	7.40	3.77
	PHATT	40.7	60.5

Table 1: Summary of run times (in milliseconds).

ACL's timing facility cannot resolve times of less than ten msec. This resulted in the system reporting a runtime of zero milliseconds for several cases in the Total and First orderings. In cases of a zero reported runtime, we have instead depicted a runtime of five milliseconds. This was done to more accurately reflect the reality of the process taking some non-zero amount of time.

Across all tests Yappr outperformed PHATT (in some cases by almost an order of magnitude). These results clearly demonstrate the gains available by using the Yappr approach. We discuss the results for the individual tests in turn.

As we described in the previous section, Yappr saves the most on its copy operations when the frontier has the fewest

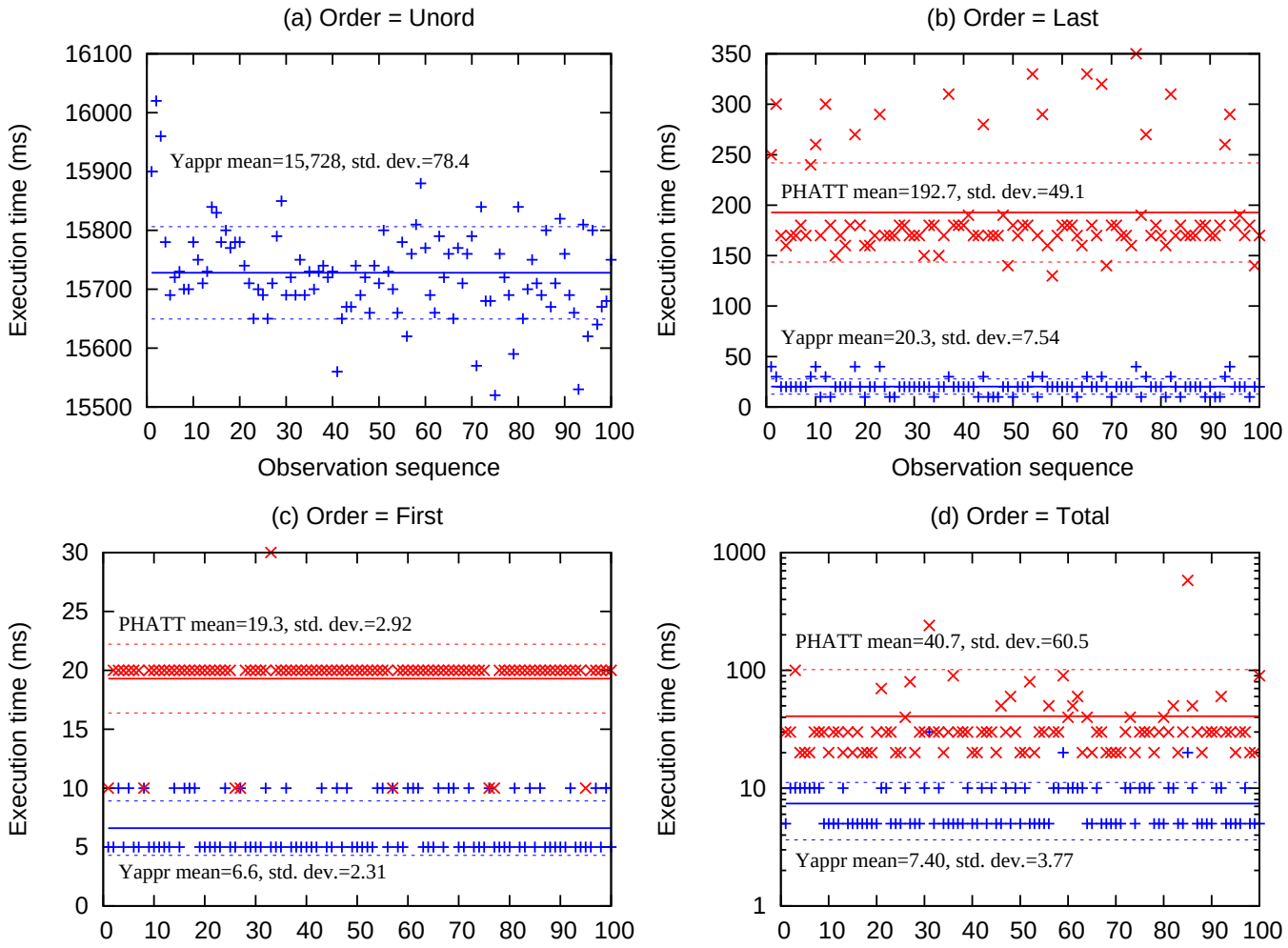

Figure 1: Runtimes comparing Yappr and PHATT. All graphs plot time in milliseconds on the vertical axis. Each Yappr and PHATT runtime is plotted as a plus-sign or X respectively. Solid lines show mean runtime with dotted lines show one standard deviation from the mean. Note in Graph (a) only Yappr's performance is shown. PHATT did not terminate on examples from this library, even allowing timeouts of several minutes.

ordering constraints. This can be seen clearly in Figure 1(a) which graphs the runtimes for plans with the Order factor = Unord. In these cases, even though PHATT and Yappr must compute the same number of explanations, the savings from the smaller models allows Yappr to solve problems PHATT is simply incapable of. In none of these cases was PHATT able to solve the problem within the thirty second timeout bound, or indeed within larger bounds of several minutes, while Yappr was able to solve all of them in under seventeen seconds. While this difference is not as profound in more ordered cases, it is present across all of the values of Order.

Consider the Order = Last case in Figure 1(b). Here Yappr outperforms PHATT by almost an order of magnitude. For clarity, in all of these figures we have added a line representing the average runtime for each system. Note that Yappr has a lower variance than PHATT. The relatively high variance of both systems is caused by considering models with multiple instances of the root goals that are discarded when the final action is seen.

Next, consider the Order = first case in Figure 1(c). While we see a drop in runtime below one second for both systems, Yappr's runtime still is less than half that of PHATT, and displays a smaller variance than PHATT. Finally, in totally ordered libraries shown in Figure 1(d) we see the same trends. Note that in this final case the runtimes on the y-axis is plotted on a log scale to accommodate the large standard deviation of the PHATT runtimes. Yappr again outperforms PHATT and has a smaller variance.

All of these experiments provide confirmation of our claims for the Yappr approach. Across the tested ordering constraints the experiments show the Yappr approach yields faster results and lower variance.

Limitations

Yappr's algorithm is optimized to be able to answer queries about the root goals being pursued in a set of plans. The model building process only returns the current explanation frontier and the set of rules used to create it. From this we can query for the root goals, but not more complex queries.

For example, suppose we want to know the probability that a particular method was used to achieve a goal. With PHATT's complete plan tree representations it would be relatively easy to select those explanations that had this structure by walking the plan trees of each of the explanations. In order to do the same thing with Yappr, the full plan structures must be reconstructed from the set of observations and the list of applied rules.

Given access to the generating trees for the original grammar, reconstructing a single explanation with n observations and m being the length of the largest plan will take $O(nlog(m))$ time. This follows from our previous argument about the complexity of explanation building. Of course, again the critical question is how many explanations need to be reconstituted. We can imagine domains where there is very little copying of the explanations and very few explanations are pruned as being inconsistent with the observations. In such domains, if these more complex queries are required, it remains to be seen if Yappr would still outperform PHATT.

Yappr is only able to recognize the set of plans that are encoded in the plan library that is initially provided. However, this does not mean Yappr is unable to generate an explanation for observation streams that contain such "unknown" plans. To do this, we include productions in the plan library that allow each action to be done as a root goal. This allows Yappr to build explanations in these cases, while still treating these more complex explanations as less likely than explanations with fewer root goals for known plans.

Related Work

We have already discussed the relationship of Yappr to the PHATT system, but there is a large amount of other related work. We are not the first to suggest that plan recognition can be done by a process similar to probabilistic parsing. Pynadath (Pynadath & Wellman 2000) proposed the use of probabilistic context free grammars for plan recognition. Yappr addresses a number of issues that are not addressed by Pynadath including cases of partially ordered plans and multiple interleaved plans.

The idea of maintaining only a subset of parse trees has been used before in parsing. Most common parsing algorithms for context free grammars, including CKY (Younger 1967), do not maintain an entire parse tree but instead only maintain the derived non-terminals of the grammar. However, to the best of our knowledge, this approach has never been used in plan recognition to reduce the overhead of maintaining explanatory models.

Conclusions

In this paper, we have argued that explicitly representing tree-based explanation in plan recognition is needlessly costly. Instead of using tree fragments and adjunction to construct models for plan recognition, we have formalized the idea of plan frontier fragment grammars and shown how they can be used to build models in a manner similar to string rewriting. We have then provided a complexity argument for this approach and shown its improved performance over the PHATT system.

Acknowledgments

This material is based upon work supported by (DARPA/U.S. Air Force) under Contracts FA8650-06-C-7606 and FA8750-07-C00017 and the EU Cognitive Systems project PACO-PLUS (FP6- 2004-IST-4-027657) funded by the European Commission.

References

Aho, A. V., and Ullman, J. D. 1992. *Foundations of Computer Science*. New York, NY: W.H. Freeman/Computer Science Press.

Avrahami-Zilberbrand, D., and Kaminka, G. A. 2005. Fast and complete symbolic plan recognition. In *Proceedings of the International Joint Conference on Artificial Intelligence*.

Blaylock, N. 2005. *Toward Tractable Agent-based Dialogue*. Ph.D. Dissertation, University of Rochester.

Bui, H. H.; Venkatesh, S.; and West, G. 2000. Policy recognition in the abstract hidden markov model. In *Technical Report 4/2000 School of Computer Science, Curtin University of Technology*.

Geib, C., and Goldman, R. 2003. Recognizing plan/goal abandonment. In *Proceedings of IJCAI 2003*, 1515–1517.

Geib, C. 2004. Assessing the complexity of plan recognition. In *Proceedings of AAAI-2004*, 507–512.

Ghallab, M.; Nau, D.; and Traverso, P. 2004. *Automated Planning: Theory and Practice*. Morgan Kaufmann.

Kautz, H. A. 1991. A formal theory of plan recognition and its implementation. In Allen, J. F.; Kautz, H. A.; Pelavin, R. N.; and Tenenberg, J. D., eds., *Reasoning About Plans*. Morgan Kaufmann. chapter 2.

Pynadath, D., and Wellman, M. 2000. Probabilistic state-dependent grammars for plan recognition. In *Proceedings of the 2000 Conference on Uncertainty in Artificial Intelligence*, 507–514.

Steedman, M. 2000. *The Syntactic Process*. MIT Press.

Younger, D. 1967. Recognition and parsing of context-free languages in time n^3. *Information and Control* 2(10):189–208.