Fast and Complete Symbolic Plan Recognition: Allowing for Duration, Interleaved Execution, and Lossy Observations

Dorit Avrahami-Zilberbrand and **Gal A. Kaminka** and **Hila Zarosim**
Computer Science Department Bar Ilan University, Israel
{avrahad1,galk,}@cs.biu.ac.il

Abstract

It is important for agents to model other agents' unobserved plans and goals, based on their observable actions. This process of modeling others based on observations is known as plan-recognition. Plan recognition has been studied for many years. It often takes the form of matching observations of an agent's actions to a plan-library, a model of possible plans selected by the agent. However, there are several open key challenges in modern plan recognition: (i) handling lossy observations (where an observation or a component of an observation is intermittently lost); (ii) dealing with plan execution duration constraints; and (iii) interleaved plans (where an agent interrupts a plan for another, only to return to the first later). In this paper, we present efficient algorithms that address these challenges, in the context of symbolic plan recognition. The algorithms allow (i) efficient matching of (possibly lossy) observations to a plan library; (ii) efficient computation of all recognition hypotheses consistent with the observations, subject to interleaving and duration constraints.

1 Introduction

Plan recognition [Kautz and Allen, 1986; Charniak and Goldman, 1993; Carrbery, 2001] focuses on mechanisms for recognizing the unobservable state of an agent, given observations of its interaction with its environment. The ability to perform plan recognition can be useful in a wide range of applications. Such applications include intrusion detection applications [Geib and Harp, 2004], virtual training environments [Tambe and Rosenbloom, 1995] and visual monitoring [Bui, 2003].

Previous investigations leave several challenges open (see Section 2 for details). First, many applications have complex multi-feature observations, rather than a single atomic feature. Some or all of these features may be intermittently lost due to noise or sensory failures (e.g., the recognizer may observe the position but not heading of an observed agent, or may suddenly lose both, for a short time). However, existing work typically assumes all features to be always observable, with no intermittent failures, and fail catastrophically if observations are suddenly missing.

Second, previous investigations typically do not utilize information on the execution duration of plans. It is possible to explicitly reason about time, and thus for example demand minimum and maximum durations for each plan. This information can be used to rule out hypotheses that match instantaneous observations, but whose hypothesized duration does not match observations over time.

Third, most previous investigations are not capable of coping with agents that pursue multiple goals. The models consider multiple goals only in sequence, where the observed agent finishes a series of plan-steps in order to finish one goal, and only then moves on to pursuing another goal. While sequential goals are certainly common in some domains, in many others agent can start with one goal, then move to another goal, and finally return to accomplish the first goal, from the point it has paused. One simple example of this is where we attempt to recognize the goal of a web user, who stops in the middle of navigating a web page and jumps to a news site, only to return to her previous work afterwards.

An ideal plan recognition system would be able to address the deficiencies above, while taking into account that observations are not just atomic instantaneous actions. But complex multi-featured tuples, involving symbolic, discrete, and continuous components (e.g., multiple actuators of the agent). The computational cost of matching such observations against all possible plan-steps is non-trivial, and also should be taken into account. However, most existing investigations ignore this cost.

This paper address these challenges, in the context of a set of algorithms for symbolic plan recognition [Avrahami-Zilberbrand and Kaminka, 2005], where the system generates plan recognition hypotheses consistent with the observations, with no ordering. Symbolic plan recognition can be very efficient, and may thus serve as a basis for a hybrid symbolic-probabilistic recognizer, where it would be useful for ruling out hypotheses prior to a more computationally-intense probabilistic reasoning process.

Specifically, this paper makes the following contributions. First, we develop a method for automatically generating a decision-tree that efficiently matches multi-feature *lossy* observations to the plan library. Second, we provide algorithms for efficiently answering the query of what is the current internal state of the observed robot (called *current state query*). While, utilizing the durations of plans, and allowing for interleaved plans.

The recognition algorithms we develop follow in the footsteps of [Avrahami-Zilberbrand and Kaminka, 2005] in their focus on completeness and efficiency. They rely on lazy commitment to hypotheses, to avoid computation of hypotheses with every step (as other algorithms do, e.g., [Geib and Harp, 2004]). Instead, they use linear-time bookkeeping with every observation, which allows extraction of hypotheses only as needed.

2 Background and Related Work

There has been considerable research exploring plan recognition. Here we only address those efforts that relate directly to the challenges addressed in this paper. YOYO* [Kaminka *et al.*, 2002] is a probabilistic plan recognition algorithm for multi-agent overhearing. The plan-library used by YOYO* included information about the average duration of plan steps, which is used to calculate the liklihood of an agent terminating one step and selecting another without being observed to do so. In this, YOYO* addressed missing observations (though their liklihood of becoming lost is to be provided a-priori). However, in contrast to our work, YOYO* did not address matching multi-feature observations (where some features may be intermittently lost), not did it allow for interleaved plans.

[Geib and Harp, 2004] describe a hybrid symbolic-probabilistic plan reconition system, allowing for interleaved plans as well as some partial observability. However, the system does not allow for taking durations into account, nor addresses efficient matching of (lossy) multi-feature observations.

There has been recent work on using hidden semi-Markov models (HSMMs) in recognizing plans [Duong *et al.*, 2005]. Hidden semi-markov models allow for providing some probabilistic constraints over the duration of plans, as well as the ability to detect anomalies. However, the model does not allow for interleaved activities, nor does it address matching with multi-feature observations.

3 Fast and Complete Symbolic Plan Recognition: The Basics

We focus on a specific model of symbolic plan recognition, briefly described below. The reader is referred to [Avrahami-Zilberbrand and Kaminka, 2005] for details.

The plan library is a single-root directed acyclic connected graph, where vertices denote *plan steps*, and edges can be of two types: vertical edges decompose plan steps into substeps, and sequential edges specify the expected temporal order of execution. Each plan has an associated set of conditions on observable features of the agent and its actions. When these conditions hold, the observations are said to match the plan. At any given time, the observed agent is assumed to be executing a *plan decomposition path*, root-to-leaf through decomposition edges.

Figure 1 shows an example portion of a plan library, inspired by the plan hierarchies of RoboCup soccer teams (e.g.

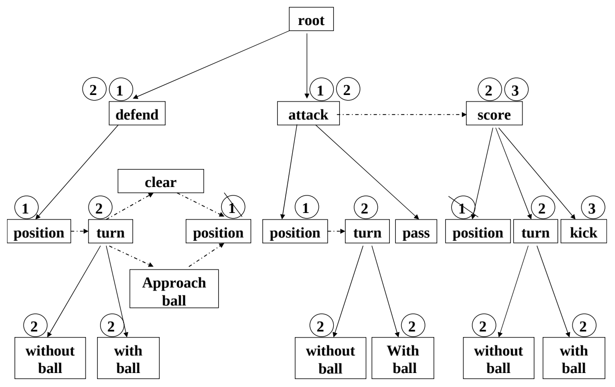

Figure 1: **Example plan library. Circled numbers denote timestamps (Section 3).**

[Kaminka and Tambe, 2000]). An observed agent is assumed to change its internal state in two ways. First, it may follow a sequential edge to the next plan step. Second, it may reactively interrupt plan execution at any time, and select a new (first) plan (we later address the case of interleaving, where the agent may resume an interrupted plan sequence).

The recognizer operates as follow: first, it matches observations to specific plan steps in the library, by using a specialized data-structure, a matching decision-tree called FDT (Feature Decision Tree). The FDT is generated automatically once prior to execution, and it efficiently matches multi-feature observations, to the plan library. Then, after matching plan steps are found, they are tagged by the time-stamp of the observation. These tags are then propagated up the plan library (see below for explanation of the $propagateUp$ algorithm), so that complete plan-paths (root to leaf) are tagged to indicate they constitute hypotheses as to the internal state of the observed agent when the observations were made.

The inference process is done by the *CSQ* (Current State Query) algorithm. The CSQ algorithm tags matching plan steps with time-stamps, and try to propagate up these tags along the plan library, so that complete paths (root to leaf) are tagged. If the propagation fails due to temporal constraints (see below), it deletes all tags it has generated in climbing up the graph.

The propagation up process done by the $propagateUp$ algorithm (algorithm 1), which tags paths in the plan library as consistent with the current observation. To do this, it must propagates these tags up along decomposition edges. However, the propagation process is not a simple matter of following from child to parent. A plan may match the current observation, yet be *temporally inconsistent*, when a history of observations is considered.

Figure 1 shows the process in action (the circled numbers in the figure denote the time-stamps). Assume that the matching algorithm matches at time $t = 1$ the multiple instances of the $position$ plan. At time $t = 1$, Propagate begins with the four $position$ instances. It immediately fails to tag the instance that follows $clear$ and $approachball$, since these were not tagged at $t = 0$. The $position$ instance under $score$ is initially tagged, but in propagating the tag up,

Algorithm 1 PropagateUp(Node v, Plan Library g, Time-stamp t)

```
1:  $Tagged \leftarrow \emptyset$
2:  $propagateUpSuccess \leftarrow true$
3:  $v \leftarrow w$
4:  while $v \neq root(g) \wedge propagateUpSuccess \wedge \neg tagged(v,t)$ do
5:      if $isConsistent(v,g,t)$ then
6:          $Tagged \leftarrow tagged \cup \{v\}$
7:          $v \leftarrow parent(v)$
8:          $propagateUpSuccess \leftarrow true$
9:      else
10:         $propagateUpSuccess \leftarrow false$
11: if $\neg propagateUpSuccess$ then
12:     for all $a \in Tagged$ do
13:         $delete\_tag(a,t)$
```

the parent $score$ fails, because it follows $attack$, and $attack$ is not tagged $t = 0$. Therefore, all tags $t = 1$ will be removed from $score$ and its child $position$. The two remaining instances successfully tag up and down, and result in possible hypotheses $root \rightarrow defend \rightarrow position$ and $root \rightarrow attack \rightarrow position$.

To disqualify hypotheses that are inconsistent (e.g., given a history of observations), it calls the algorithm *isConsistent* (2) to make the decision of whether a proposed time-stamp should be applied to a given plan step, given information in the model. It assumes that the calls to it have been made in order of increasing depth, from the parent to the children, to avoid the case that the children are tagged, but their parent does not

In [Avrahami-Zilberbrand and Kaminka, 2005], a version of *isConsistent* algorithm is presented which checks for temporal consistency. It is repeated here in Algorithm 2. Line 2 checks whether time stamp t is temporally consistent, i.e., if one of three cases holds: (a) the node in question was tagged at time $t-1$ (i.e., it is continuing in a self-cycle); or (b) the node follows a sequential edge from a plan that was successfully tagged at time $t-1$; or (c) the node is a first child (there is no sequential edge leading into it). A first child may be selected at any time (e.g., if another plan was interrupted). If neither of these cases is applicable, then the node is not part of a temporally-consistent hypothesis, and its tag should be deleted, along with all tags that it has generated in climbing up the graph. This final deletion of all failing tags takes place in the CSQ Algorithm. The CSQ is meant to be called with every new observation. The tags made on the plan-library are used to save information from one run to the next.

In sections 4.1 and 4.2, we modify this consistency check and extend it, so that it takes additional constraints into account.

4 Accounting for Complex Temporal Plan

The basic model described above may be used to recognize plan(s) that are ordered in time. It also allows for plans to have self-cycles, and thus be non-instantaneous. However, it cannot recognize more complex forms of temporal plans, such as maintaining the selection of a specific plan-step within some bounded interval, or interrupting a sequence of plan steps under one node, to execute another, only to return to it to the first sequence later (plan interleaving).

Algorithm 2 isConsistent(Node v, Plan Library g, Time-stamp t)

```
1: if $tagged(parent(v),t) \vee features(parent(v)) = \emptyset$ then
2:     if $tagged(v,t-1) \vee \exists PreviousSeqEdgeTaggedWith(v,t-1) \vee NoSeqEdges(v)$ then
3:         return $true$
4: return $false$
```

This section shows how the CSQ mechanisms can be extended to account for these temporal plans. Section 4.1 extends the $propagateUp$ algorithm to take duration bounds (minimum and maximum time spent in a plan-step) into account. Section 4.2 independently explores modifications to $propagateUp$ for interleaving.

4.1 Managing Durations

Instances of the same plan step can vary in the duration of their execution. For example, depending on the distance to the ball, a soccer player may take a long time or short time to execute the *approach ball* plan in Figure 1. As a result, we may have multiple observation time-stamps ($t, t+1, \ldots t+k$) that are all consistent with a single plan, and only reflect the duration that its execution requires between one and $k+1$ time-stamps.

However, often some bounds are known on execution duration. For instance, in an airport terminal, there exist a difference in the plans of a a passenger who stands at the check-in area for a few minutes, and a security guard who stands there for a few hours. Or, in a different example, a basketball player is only allowed inside the basket zone for a limited amount of time.

We thus want to take into account constraints on the duration of plan-steps. We can allow such constrains in the approach we presented. We extend the tagging mechanism to allow two types of tags: (a) *Hard tags*, which signify that a plan step is (or is not) consistent with respect to previous plan-steps (the $isConsistent$ algorithm, Algorithm 2), and also consistent with duration constraints (minimum, maximum time); and (b) *Soft tags*, which signify that a plan-step is (or is not) consistent with observations (and with prior plan-steps), but is not within its duration constraints (e.g., this plan-step has not been selected for sufficient amount of time, but may be in few time-stamps).

We make the following additions to the $propagateUp$ algorithm, and present the revised algorithm (Algorithm 3): (1) Line 4 checks if v is not tagged with *hard tag* (instead of checking if tagged); (2) line 5: checks if the maximum duration holds, using $calcDuration$ algorithm (Algorithm 4); (3) Line 6: in the $isConsistent$ algorithm, all the occurrences of $tagged(v,t)$ should be replaced with $tagged(v,t,Soft)$, and the $existsPreviousSeqEdgeTaggedWith(v,t-1)$ should be

Algorithm 3 PropagateUp(Node v, Plan Library g, Time-stamp t)

```
1: $Tagged \leftarrow \emptyset$
2: $propagateUpSuccess \leftarrow true$
3: $v \leftarrow w$
4: while $v \neq root(g) \wedge propagateUpSuccess \wedge \neg tagged(v, t, Hard)$ do
5:   if $calcDuration(v, t) < maxDuration(v)$ then
6:     if $isConsistent(v, g, t)$ then
7:       if $calcDuration(v, t) < minDuration(v) - 1$ then
8:         $tag(v, t, Soft)$
9:       else
10:        $tagDuration(v, t, Hard)$
11:      $Tagged \leftarrow tagged \cup \{v\}$
12:      $v \leftarrow parent(v)$
13:      $propagateUpSuccess \leftarrow true$
14:    else
15:      $propagateUpSuccess \leftarrow false$
16:  else
17:    $propagateUpSuccess \leftarrow false$
18: if $\neg propagateUpSuccess$ then
19:   for all $a \in Tagged$ do
20:     $delete\_tag(a, t)$
```

Algorithm 4 calcDuration(Node v, Time-stamp t)

```
1: $counter \leftarrow 0$
2: $time \leftarrow t$
3: while $time > 0$ do
4:   if $tagged(v, t-1, Soft)$ then
5:     $Counter \leftarrow$counter+1
6:     $time \leftarrow$time-1
7:   else
8:     $break$
9: $return$(counter)
```

replaced with $existsPreviousSeqEdgeTaggedWith(v, t-1, Hard)$.; (4) lines 7-12 if the minimum duration constrain holds, then the plan will be tagged in time-stamp t with $Hard$ tag, otherwise with $Soft$ tag.

The algorithm $calcDuration$ (Algorithm 4), calculates the duration in which the plan was active. To calculate this, the algorithm goes over the time-stamps of the plan from time stamp t, and counts the number of consecutive tags (time-stamps with no temporal gaps). For example, if plan is tagged with time-stamps 1,4,5,6 then the duration will be 3.

4.2 Interleaved plans

Many plan recognition algorithms cannot cope with modelling an agent that is pursuing multiple plans (i.e., for multiple goals), by interleaving plan steps. Here, the agent may begin with one plan, and interrupt its execution to execute another, only to return to the remaining plan steps in the first plan.

To handle interleaving, we add a $memoryFlag$ in each of the first children. this flag will hold the latest time-stamp tag, in the sequential link chain from this child. We will use this flag to disqualify plans that are in the middle of the chain and are not the ones that we paused at. For example, in the 1 figure, the position plan under attack will hold $memoryFlag$ that contain the time-stamp 2. Suppose that there is sequential link also from turn to pass. Then we would like to return exactly to pass and not to the turn plan step.

The approach we present above can deal with these cases by making one change to the $isConsistent$ algorithm (algorithm 2). In line 2, the condition: $\exists IncomingSeqEdgeTagged(v, t - 1)$ should be replaced with two conditions: First, $\exists IncomingSeqEdgeTagged(v, t - 1 or smaller))$. Meaning that we check whether there is an incoming sequential edge from a plan-step tagged with a time-stamp that is smaller or equal to $t - 1$. This will allow a plan-step to be considered consistent if it continues a previously interrupted sequence of plan-steps. Second, we add the condition $\exists IncomingSeqEdgeTagged(v, MemoryFlag)$, this condition forces a return to the interrupted plan-step.

Although we add recognition capabilities, we should remember that this change can influence the number of possible hypotheses. A partial solution is to tag those plans that can be interrupted and resumed with a $jump$ label. The $isConsistent$ algorithm will then check plans based on their $jump$ settings. For plans with a $jump$ label, it will check that the previous node had time equal to $t - 1$. For plans without a $jump$ label, it will check equal or smaller than $t - 1$ constraint. This allows greater control over the accuracy of the model, and facilitates increased efficiency.

Another issue to discuss is the question of how many ticks (time units) can pass from the time we interrupt a plan step, until returning to it. To limit this time, we can add a flag $MaximumInterruptTime$, and add also the constraint that the difference between a new observations's time-stamp and that of a node with an incoming sequential edge, must be smaller than $MaximumInterruptTime$. This can disqualify lingering hypotheses, and make the algorithm more accurate.

5 Handling Lossy Observations

Real-world applications of plan recognition may violate the assumption that the observed agent is always observable, and that all relevant features are observable. Instead, real-world settings often involve intermittent observation failures. This section addresses this challenge. We differentiate between two types of lossy observation: (a) Lossy features, where one or more features in a multi-feature observation is temporarily missing (Section 5.1); and (b) lossy observation, where an entire observation is lost (5.2).

5.1 Lossy Features

We take each observation to consist of a tuple of observed features, including states of the world that pertain to the agent (e.g.,a soccer player's uniform number), actions taken (e.g., $kick$), and execution conditions maintained (e.g., $speed = 200$). In most applications an implicit assumption was made (present also in most related work) that all relevant features were in fact observable.

However, in realistic settings, some features may be intermittently unobservable, e.g., due to hardware failures, communication errors, etc. For instance, due to a sensor failure, a plan recognition system might only know the position of another agent, but not its velocity or heading. Observation features that are lost would fail the conditions associated with plan, and thus the matching phase will fail.

In [Avrahami-Zilberbrand and Kaminka, 2005], we showed how to efficiently determine which plans match a set of observations, using structure called FDT (Feature Decision Tree). The FDT allows efficient mapping from observations to plans that may match them, at a worst-case runtime of $O(F + l)$, where F is the number of features, and l the maximal number of plans that may match a given observation.

An FDT is constructed similarly to a machine-learning decision tree [Ross, 1992] (though with some differences). We briefly review this process here. We map the plan library into a set of fictitious training examples. Each plan step becomes an example, where the conditions on feature values become attribute values, and the class is the plan step. Features not tested by a plan step are treated as specially marked *all values*. After generating the training set, the construction of the FDT is done as by picking features in decreasing order of information gain, and constructing a decision tree which tests these features. Leaves in the decision tree contain pointers into the appropriate plan steps in the plan library (see Figure 2; the *Miss* branches are explained below).

To address lossy features, we propose to use an augmented FDT, called LFDT (Lossy Feature Decision Tree), which has the matching run-time of a non-lossy FDT, but deals with lossy observations. An LFDT representation is the same as FDT, except that for each node, we add an extra branch that represents a *missing value* (*Miss* in Figure 2). During construction of the LFDT, all plans that are consistent with the node (and which are divided based on the value of the feature associated with the node) would be passed as-is to the missing value branch. When the LFDT is traversed, if a feature is temporarily unobservable, we will follow the missing value branch instead of one of the normal branches.

To understand the LFDT construction process, we need first to differentiate between two special values that each feature can take: a *miss* value is used during the matching process to denote that the value of this feature has lost. An *all values* value is used during the FDT and LFDT construction process to denote that this feature is not relevant for the plan. i.e., the plan does not test this feature. For Example, the feature uniform number is irrelevant to the approach ball plan;

The LFDT construction algorithm is presented below (Algorithm 5). First, we check if the instances cannot be divided, meaning that a node points at only a single plan, or there are no more features that can differentiate between the plans associated with the instances. In this case we create a leaf (lines 1–2). Otherwise, we create a node, and associates it with the feature that provides the greatest information gain (lines 3–4) (intuitively, that divides plans that test it as uniformly as possible). We then create children LFDT nodes for each of its values including the potential *miss value* (lines 5–12), and recursively repeat the process of selecting a feature that best divides the plans associated with the node.

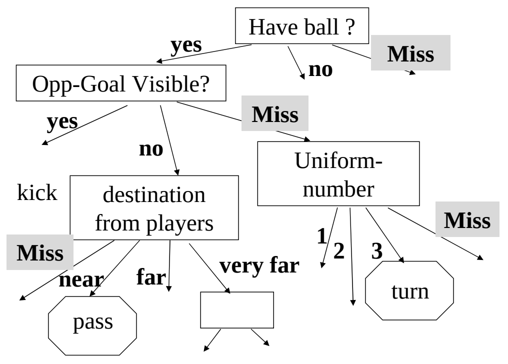

Figure 2: An example Lossy Feature decision Tree (LFDT).

To create the children LFDT, we follow the procedure for handling missing values in decision trees (See [Ross, 1992]). Briefly, the algorithm (in each step), divides the instances according to the tested features. Each instance gets a weight, that is initialized to one at the beginning of the algorithm. This weight represents the fraction of the instances having this value for the feature, and is used for the purpose of computing the information gain. When there is $allvalues$ value for the tested feature in the instance, we divide this weight between all branches, therefore dividing its weight in the number of possible values that the feature can take. When having value for the feature, the weight of the instance remains the same.

The children are constructed as follows: For each possible value of the selected feature, the instances will be divided to the different branches and the weights will be updated in the following manner: if this is *miss value* (represents "missing value" branch) then all instances remain as its parent (line 7), and the weights also remain the same (line 8). This child is different from its parent only in the best features that it can select (line 12). If this is not *miss value*, then the instances will be divided to the different branches according to their values with the same weights. In case this is $allvalues$ value the instance will be passed to all branches". And its weight will be divided in the number of the values of this feature. The algorithm also update the $Tested$ set with the new tested feature. Then we recursively repeat on this process of selecting a feature that best divides the plans associated with the node with the new instances, new weights and the new tested features, dividing accordingly.

Thus the construction of the LFDT is done as in the FDT, but with one small difference. For a missing value node, the plan step examples remain as in its parent, and the weights also remain the same. This missing value node differs only by the set of features from which it selects (as it represents a feature that was missing). This process guarantees that all plan-steps that are consistent with the partial observations will be matched, even if some features are intermittently unobservable.

Complexity Analysis. During matching runtime, the LFDT

Algorithm 5 formTree($Instances, weights, Tested$)

```
1:  if (there are no features to test) $\vee$ (single plan) then
2:      return $createLeaf(Instances)$
3:  $bestFeature \leftarrow$ best feature that was not tested
4:  $createNode(bestFeature)$
5:  for all possible values $v$ of best feature do
6:      if $v =$ missing value then
7:          $newInstances \leftarrow Instances$
8:          $newWeights \leftarrow Weights$
9:      else
10:         $newInstances \leftarrow$ all instances with value $v$
11:         $newWeights \leftarrow$ calculate weights of $newInstances$
12:     $newTested \leftarrow Tested \cup bestFeature$
13:     $formTree(newInstances, newWeights, newTested)$
```

Match algorithm operates as follows: When an observation is made of an agent, we traverse on the LFDT according to the values of the observed features. It the feature is observed, its value is used as an index to select from one of the normal branches. However, if the value is *miss* (meaning that the feature's value is unavailable), we take the *missing value* branch. This process continues recursively, until we get to a leaf. Then, after getting to the appropriate node in the LFDT, we have pointers to the relevant plans in the plan library, the same as we have in the FDT. These pointers are returned.

In a theoretical worst case, plans test all possible features, and thus the height of the LFDT is $O(F)$. In the worst-case, the leaf in the LFDT would point to $O(l)$ plans, where l is the maximum number of plans that are ambiguously consistent with a single observation ($l << L$, where L is the number of plans in the library). Thus the complexity of matching observations to plans would be at worst $O(F + l)$, similarly to the FDT.

As with any decision tree, there is a one-time computational cost of constructing the LFDT, and storage overhead in using it. The construction complexity of the LFDT is $O(FVL(V + 1)^{F+1} + l)$, where F, l are defined as above, and V is the maximum number of values in each feature, and L is the number of different plans in the plan library. Since for each node ($O(V + 1)^{F+1}$), we go over all features to decide which feature is the best to choose, by computing the information gain ($O(FVL)$). However, building the LFDT is a one-time offline cost, while matching takes place many times in realistic settings.

While the runtime complexity of LFDT is the same as FDT, its storage complexity would be greater, as it will have more branches than FDT (extra branch for each feature), and its height may be deeper than FDT (because of the need to handle missing features at the leaves).

The space complexity of the LFDT is $O((V + 1)^{F+1} + l)$ in the worst case, where V is the number of values for each feature (full tree whose height is $F + 1$ and whose branching factor is $V + 1$). To lower the size of the LFDT in practice, we can add an extra branch just to lossy features, i.e., not in all features, but only in features we know the observing agent may miss.

Trading Time for Space. Given the increased space complexity of the LFDT, we develop an alternative way to deal with lossy features. Here, we use the FDT while accounting for missing values. The difference is in the matching algorithm. Instead of turning to the *missing value* branch, if we do not have the observed feature value, we will traverse all branches of this feature, collecting the pointers resulting from each.

To demonstrate the lossy matching process with FDT, lets take the example in Figure 2 without the "miss" branches. Assume that the observation is: the value of the $have - ball$ feature is yes, the value of the $opp - goal - visible$ feature is no, and the value of the $destinationfromplayers$ is unobservable. In this case we will check all the branches under the feature $destinationfromplayers$, instead of taking the missing value branch in the LFDT. We will traverse the branches: near, far and very far, according to the observation, collecting the pointers into the plan library, resulting from each such branch.

The matching runtime complexity is then changed from $O(F + l)$ in the worst case, to $O((V + 1)^{m+1} + l)$, where m is the number of missing features. It depends now both on the number of missing features in the observation and on the number of values in each feature. However, it is trade off between the space complexity and the matching runtime complexity.

5.2 Missing Observations

An underlying assumption in previous investigations is that every change in internal state (in our terms, change in plan path) is somehow reflected in observations. However, in realistic settings, this assumption is sometimes violated (e.g., in Overhearing applications [Kaminka *et al.*, 2002]. Some internal decision-making may be permanently or intermittently unobservable, for all of the plans along a specific plan decomposition path. In this case, an entire observation is essentially missing (all features are unobservable).

We propose some small changes in the propagation algorithms, to allow them to address this difficulty. The idea is to mark potentially-unobservable plans with a *lossy* label in the plan library (though the first and the last plans in a plan-step sequence cannot have a *lossy* label).

We again modify the algorithms $isConsistent$ (Algorithm 2) and $propagateUp$ (Algorithm 1). In the propagating process, nodes that are labelled with *lossy* and are part of a sequence will be skipped (if they are not tagged), when the sequence is checked for temporal consistency. This is done by replacing the method $tagged(X, t)$ with a new method: $AdvanceTagged(x, t)$ (Algorithm 6).

The $AdvanceTagged(v, t)$ algorithm exploits the sequential edges and the *lossy* labels. Plan v will be considered as tagged at time-stamp t if one of two cases holds: (a) the plan itself is tagged with time-stamp $t-1$; (b) the plan is not tagged with time-stamp $t-1$, but it has *lossy* label and the sequential edge that points to it is tagged with time-stamp $t - 1$. In case that there are chains of *lossy* labels, we check if the first plan that is not labelled with *lossy* is tagged with time-stamp $t-1$.

It is important to note here that the time referred to by the time-stamp is the observation time, not world time. Thus $t-1$

Algorithm 6 AdvanceTagged(Node v,Timestamp t, Plan Library g)

```
1: if $tagged(v,t)$ then
2:     return true
3: else
4:     while ($v$ labelled as miss) $\wedge$ ($\exists X$, s.t. $(X,v) \in g$) do
5:         if $tagged(X,t)$ then
6:             return true
7:         $v \leftarrow X$
8: return false
```

is the time of the previous observation, not a single tick ago.

This feature must be used carefully, since it can significantly influence runtime, and the number of possible hypotheses. Specifically, labelling many plan steps as *lossy* will result in long backtracks through previous plan-step nodes, until we arrive at node that was not labelled with *lossy* or that was tagged with time stamp $t-1$. This can be significantly more expensive to do than just checking whether the previous node on the sequence was tagged with time stamp $t-1$. The number of output hypotheses also increases when adding many *lossy* labels, because a behavior labeled *lossy* can be a part of many hypotheses, without being observed.

6 Summary and Future Work

Agents must often rely on their observations of others, to infer their unobservable internal state, such as goals, plans, or selected plans. However, plan-recognition approaches to this task leaves number of open challenges: (i) handling lossy observations; (ii) dealing with durations constrains ;(iii) facing interleaved plans (behaviors).

This paper addresses this challenges in the context of symbolic plan recognition, relaying on hierarchical plan-based representation and a comprehensive set of algorithms that can answer a variety of recognition queries. The algorithms we propose are efficient, and can handle intermittent failures in observations, plans with duration, and lossy observations, both of complete observations and of only a subset of observable features.

References

[Avrahami-Zilberbrand and Kaminka, 2005] Dorit. Avrahami-Zilberbrand and Gal A. Kaminka. Fast and complete symbolic plan recognition. In *IJCAI-05*, Scotland, Edinburgh, 2005.

[Bui, 2003] H. Bui. A general model for online probabilistic plan recognition. In *IJCAI-03*, 2003.

[Carrbery, 2001] S. Carrbery. Techniques for plan recognition. *User Modeling and User-Adapted Interaction*, 11:31–48, 2001.

[Charniak and Goldman, 1993] Eugene Charniak and Robert P. Goldman. A Bayesian model of plan recognition. *AIJ*, 64(1):53–79, November 1993.

[Duong *et al.*, 2005] T. Duong, H. H. Bui, D. Phung, and S. Venkatesh. Activity recognition and abnormality detection with the switching hidden semi-markov models. In *IEEE International Conference on Computer Vision and Pattern Recognition (CVPR-2005)*, San Diego, CA, June 2005.

[Geib and Harp, 2004] Christopher W. Geib and Steven A. Harp. Empirical analysis of a probalistic task tracking algorithm. In *AAMAS workshop on Modeling Other agents from Observations (MOO-04)*, 2004.

[Kaminka and Tambe, 2000] Gal A. Kaminka and Milind Tambe. Robust multi-agent teams via socially-attentive monitoring. *JAIR*, 12:105–147, 2000.

[Kaminka *et al.*, 2002] Gal A. Kaminka, David V. Pynadath, and Milind Tambe. Monitoring teams by overhearing: A multi-agent plan recognition approach. *Journal of Artificial Intelligence Research*, 17, 2002.

[Kautz and Allen, 1986] Henry A. Kautz and James F. Allen. Generalized plan recognition. In *AAAI-86*, pages 32–37. AAAI press, 1986.

[Ross, 1992] Quinlan J. Ross. *C4.5 Programs for machine learning*. Morgan Kaufmann Publishers,Inc, 1992.

[Tambe and Rosenbloom, 1995] M. Tambe and P. S. Rosenbloom. RESC: An approach to agent tracking in a real-time, dynamic environment. In *IJCAI-95*, August 1995.