Bar-Ilan University
Department of Computer Science

SYMBOLIC BEHAVIOR RECOGNITION

by

Dorit Avrahami

Submitted in partial fulfillment of the requirements for the Master's degree
in the department of Computer Science

Ramat-Gan, Israel
September 2004

This work was carried out under the supervision of

Dr. Gal A. Kaminka

Department of Computer Science, Bar-Ilan University.

Abstract

It is important for robots to model other robots' unobserved plans, goals and behaviors, based on their observable actions. This process of modeling others based on observations is known as behavior- or plan-recognition. Behavior-recognition algorithms work by first matching observed actions to a template model (called the plan- or behavior-library), and then propagating the implications of matching actions to determine possible hypotheses that explain the observed behavior. However, classic plan recognition algorithms are ill-suited to modeling robots in state-of-the-art applications: (i) they assume that only a single atomic feature (i.e., the action of the observed robot) is observable at any given point; and (ii) they assume that all such actions are always observable (i.e., the observer never loses an observation). As a result, existing behavior-recognition algorithms are often inefficient, and may fail catastrophically in face of lossy observation streams.

This thesis presents a set of behavior-recognition algorithms that are specifically suited for modeling behavior-based robots. First, the algorithms use a decision-tree structure to efficiently match complex (multi-feature) observations to behaviors, reducing the run-time complexity of the observation-matching phase from $O(FL)$ to $O(F + L)$ in the worst case. Second, the algorithms are able to handle lossy observations gracefully. The algorithms are correct (in that all matching hypotheses are produced), and symbolic (in that they do not provide an ordering over hypotheses). The algorithms' run-time is linear in the size of the behavior-library. We provide an extensive empirical evaluation of the algorithms in scaled-up simulation experiments.

Acknowledgments

I would like to express my deep gratitude to Dr. Gal A. Kaminka for his support and understanding during the course of research. For believing in me right from the first beginning of the research. This work would not have established without him.

Special gratitude to Maverick group for their support and encouragement.

Further, I wish to express my gratitude to my husband Nadav and to my family, for standing by me throughout the whole way, my parents, Magid and Gila, my sister Ronit and my brother Tomer.

Finally, I want to express my gratitude to Batya and Menahem Zilberbrand for their support.

Contents

List of Figures

List of Tables

Chapter 1

Introduction

It is important for agents to be able to reason about other agents' internal state, such as their selected behaviors, plans, intentions, and goals. A model of other agents is important, for instance, in assisting other agents [22], countering their adversarial actions [38], imitating them [2] and detecting failures in multi-agents environments [19, 18]. Since it is often impractical for a agent to rely on its peers to continuously transmit their internal unobservable state to it, an agent modeling its peers must often rely on inferring its peers' unobservable state based on their observable actions.

The problem of inferring another agent's intentions based on set of observations is often called *plan recognition*. Plan recognition is useful in many areas, some applications are natural language question answering systems and story understanding (e.g.,[1, 25, 39]), intelligent user interfaces (e.g.,[12]), automated description of image sequences [31], and multi agent coordination (e.g.,[16, 15, 17, 19, 18]). Although much research has dealt with plan recognition, there are number of problems when dealing with large scale of behaviors and real world applications [4].

Most plan recognition methods are ill-suited to modeling modern robots. First, plan recognition typically assumes that observed actions are atomic and instantaneous. However, robots often take continuous actions that have duration, and are complex (multi-featured), as they affect several actuators at once (e.g., maintain velocity and direction over a period of time). Second, plan recognition focuses

on observed actions, and ignores how world state may affect the observed robot's state (e.g., a robotic soccer player's internal decision making will be affected by its position on the field, or its uniform number). Third, the representation underlying plan-recognition is based on STRIPS-like operators, a representation not commonly used in generating reactive behavior in robots. A plan-based model of a robot's interaction with the environment would not capture the reactive components of its behavior. Fourth, plan-recognition typically focus on small class of agent behaviors (plan-based), as seen in static, single agent domains.

There have been attempts at addressing these challenges (see Section 2 for details). RESC [38] and RESL [19] use a behavior-based representation to infer the current behaviors selected by observed agents. However, they do not take a history of observations into account, and assume that agents do not change states (behaviors) unobservably. Other methods are often able to take a history of observations into account, but assume all relevant features (e.g., actions of the robot) will always be observable (e.g., [16, 30, 13]). Moreover, these methods require a translation of the observed robots behavior-based control structure into a form suitable for the probabilistic recognition algorithms used in these approaches. Also, none of the approaches discussed above can utilize negative evidence, i.e., inference from a lack of an observation [11].

This thesis focuses on a set of comprehensive mechanisms for *behavior recognition*, the task of recognizing the unobservable behavior-based state of an agent, given observations of its interaction with its environment. We examine the key behavior recognition queries that may be asked of a behavior recognition system, and provide algorithms to infer the answers to these queries, building on a a representation of hierarchical behavior that is general and compatible with many existing behavior-based control methodologies. We analyze the complexity of the algorithms, and show that they are efficient, even when handling loss of observations, and negative evidence. The algorithms are all symbolic, in that they produce all possible hypotheses that are consistent with the observations, but do not provide a probability distribution over the hypotheses space. However, their efficiency makes them suitable as a basis for additional probabilistic reasoning.

In addition, we address the efficiency of matching observations to behaviors, a key basic step common to all behavior recognition algorithms. Previous work

has assumed observed actions are atomic (have a single observable feature), and scaled up linearly in the size of the library. Instead, we develop a method for automatically generating a decision-tree that efficiently matches observations, based on the values of observed features, to appropriate behaviors. This method reduces the complexity of matching from $O(FL)$, where F is the number of observable features, and L the size of the behavior library, to $O(F + L)$ in the worst case.

In addition to the development and analysis of these algorithms, we present an extensive empirical investigation of the performance of the behavior recognition mechanisms on simulated data. The experiments show the efficacy of the proposed techniques, as well as the scope of their strengths and weaknesses.

Chapter 2

Background and Related work

Plan recognition is the task of inferring the intentions, plans, and/or goals of an agent based on observations of its actions. To demonstrate it, lets take a simple example, where we observe a person leaving her house. There are number of possibilities as to her intentions: going to work, taking out the garbage, going to the fitness club, walking with the dog and so on. Suppose that some time before, we saw her taking the car keys and her work bag. Now, we can disqualify a lot of hypotheses as to her intentions and infer that she is going to work. A lot of research has been done in the plan recognition area, [33] made psychological research that support evidence that humans infer the plans of other agents, and therefore engage in plan recognition. Behavior recognition is a specialized form of plan recognition. Here the task focuses on inferring the internally-selected behavior-based control module of another agent from a set of observations of its actions.

The recognition system can be characterized by the following properties:

1. *Keyhole* and *intended* recognition are two types that were identified by [7]. In keyhole recognition, the observed agent does not impact on the recognition process, whereas in intended recognition, the observed agent does deliberate actions to help the recognition. Another class was identified by [9], *adversarial recognition*, where the observed agent is hostile (e.g, network security) and takes steps to confuse the observer. Most of previous work investigated keyhole recognition (e.g,[21, 6, 19, 38]). We would also

focus on the keyhole recognition, where the observed agent does not impact the recognition process.

2. The recognition system needs to represent its hypotheses in some fashion. It typically holds a structure, which defines the expected relationships among goals, behaviors, plans and primitive actions that possible in the specific domain. This is often referred to as the representation of the plan-library (and in our case, behavior library), which bounds the expected behavioral repertoire of the observed agent.

   Many different representations have been proposed in previous work: Action taxonomies [21], hierarchical task network (HTN) [23], Bayesian networks[6], behavior-based control methodologies (e.g., [8, 29, 28, 26, 3]) and many others. In our work we utilize a hierarchical behavior-based recognition representation which serves as the basis for representing the modeled behaviors of the observed agent. The representation is generic, and be used to represent behavior-based controllers of various forms. We follow the bulk of earlier work in assuming the library is correct and complete with respect to the actual behaviors used by the observed agent.

3. There are various assumptions that can be made as to what the observation should include. Most Recognition systems (e.g, [21, 38, 6]) take into account just the actions taken by the agent, they do not refer to the changes in the state of the world, beliefs of the agent, and many other features that can influence on the recognition process. Moreover, many approaches assume that the sequence of observations is fully-observable, meaning all actions done by the agent can be observed, with no gaps in the sequence (e.g,[21, 19, 6, 31, 38]). However, in real world applications this is not always the case, some actions may be intermittently unobservable, e.g., due to hardware failures. In our work we consider complex observations, that consist of a tuple of observed features, including states of the world, actions taken by the agents, and execution conditions maintained by the agent. See chapter 4 for more details.

4. Taking in to account an ordered history of observations of the agent also

need to be considered when characterizing recognition systems. There are approaches that do not consider at all the history of observations (e.g, [38, 19]), whereas most approaches consider the history of observations (e.g, [5, 6, 9]). In our work we are utilize the history of observations to disqualify hypotheses.

5. There are approaches that do not consider the possible order between behaviors of agents (e.g, [38, 19, 6]), whereas approaches that do take into account the ordering constrains between behaviors (e.g,[9]). Our work uses temporal constraints to disambiguate hypotheses.

6. A plan- or behavior-recognition system may answer several different queries: (i) current state query—what is the current behavior the agent has selected now? (ii) history states query—what is the sequence of behaviors the agent has selected over time? (iii) future states query—what is the next behavior to be selected by the agent? etc. Most approaches can give an answer just to the current state query, some of them gives the answer without history consideration (e.g, [38, 19]), and others take into account the history (e.g, [9]). As to our knowledge, none of the approaches give answer to the history state query. We provide algorithms for both of these queries.

7. *Correctness and completeness.* It is likely, in realistic settings, that more than one behavior will match a set of observations, and this may result in multiple hypotheses as to the internal state of the observed agent (or sequence of internal states). Recognition algorithms are called *complete* if they return all hypotheses that match the observations, and *correct* if they return only hypotheses that match the observations. Symbolic approaches (such as ours) may be characterized based on their correctness and completeness [21, 19]. Probabilistic approaches provide a ranking of hypotheses to indicate likely useful hypotheses. Some approaches provide multiple hypotheses (with or without ranking), while others commit to a single hypotheses [38]. We choose to deal with the symbolic approach for number of reasons: First, symbolic computations are sufficient for specific tasks (such as failure detection [19]). Second, they are generally much more efficient

than probabilistic approaches, and can thus serve as a useful pre-processing step to such approaches, to limit their run-time in practice.

8. The recognition process, in most applications, is done online (in real time), and therefore the recognition algorithms must be efficient. Most algorithms in the plan recognition area use small domains, with few number of plans and goals. The problem of scalability and complexity was mentioned as problem from the early work of plan recognition [20]. Here in our work, we present algorithms that are efficient: linear in the size of the behavior tree, and can deal with large number of behavior and hypotheses to answer queries.

9. *Negative evidence.* Exploiting new events to disambiguate previous hypotheses that were considered as true. Few works has addressed to this property (e.g,[11]).

The well-known work of Kautz [21] provided a formal theory of plan recognition. In this work the problem is viewed as a deductive inference, and relies on a representation called *action taxonomy*, where every observed action is a part of one or more "top level plans". The task of the plan recognition is to find minimal set of top plans that explain the observations. Kautz's work handled many difficult cases, such as allowing for observations to come in at any order.

However, this approach faces inherent difficulties when applied to the complex, dynamic settings in which agents are typically deployed. First, agents may take continuous (servo) actions, intended to maintain some interaction with their environment (e.g., velocity or heading). The actions in this case are not discrete and instantaneous, but instead are composed of multiple continuous changes to approximate some target function [29]. Modeling such actions using discrete operators is difficult at best. Second, since the input to classic plan recognition algorithms is a stream of actions, it is difficult to incorporate additional context that may be observable, and may be affecting the internal decision-making of the observed robot. For instance, a robotic soccer player knows its own uniform number, which can be also observable to the modeling robot. Its uniform number affects its unobservable state but knowledge of the uniform number is not utilized by plan

recognition techniques. Third, agents that operate in complex, dynamic settings, may interrupt planned action sequences in order to react to unexpected situations. To do this, many robots utilize a behavior-based control approach, which executes multiple control modules (called behaviors) so as to produce the desired behavior, while still maintaining the ability to react to unexpected situations [26, 29, 3, 24].

An ideal behavior recognition system would be able to address the deficiencies above, while taking into account a history of observations to infer answers for several types of recognition queries

There have been several relevant previous investigations. RESC [38] uses a hierarchical behavior-based representation to infer the current behaviors selected by observed robots. In each run-time cycle, RESC maintains only a single hypothesis as to the current state of the observed robot. RESL [19] is similar, but maintains multiple hypotheses as to the current state. Both algorithms essentially reset with every new observation, and do not take a history of observations into account. Thus they cannot provide hypotheses as to the sequence of unobservable states that the observed agents has gone through, nor can they provide predictions as to the next possible state. Finally, RESC and RESL assume that any change in the internal state will have some observable evidence.

Other alternatives to classic plan recognition are also relevant. Many of these are probabilistic in nature, and also are able to take a history of observations into account (though they often ignore the history of internal states).

[5, 6] constructed the first Bayesian plan inference system. Their system first retrieve candidate explanations, then these explanations were inserted to the Bayesian network. In the Bayesian network, the random variables (nodes) represented propositions, the root nodes represented hypotheses about an agent's plan. Each node's probability, represents the likelihood of the proposition given the evidence provided by its parents and its children. As new evidence is added to the network, the probabilities at each node are recomputed, by propagating the evidence through the nodes. This approach has number deficiencies: First, this approach requires a large number of prior and conditional probabilities, that not always available. Second, there is no distinction between plans and actions, there is no consideration on other features, such as state of the world. Third, it is not sensitive to the order of the plans. The complexity of reasoning using this rep-

resentation is that of general Bayesian network, i.e., NP-Complete in the general case.

[16] also explores an approach in which a Bayesian network for plan-recognition

be applied to real time dynamic domains with unobservable actions, it assumes that the initial state of the world before each observation is known, can not do predictions, and finally it can not utilize the negative evidence property.

Chapter 3

Representation

Many state-of-the-art robotic controllers employ hierarchical behavior-based control methodologies (e.g., [8, 29, 28, 26, 3]). A behavior is a controller for reaching and/or maintaining a particular goal (see, for instance, [26]). For example in the robot domain, *following* is a controller that keep the robot moving within a fixed region behind another moving agent. Often, behaviors are applied in parallel, in a hierarchical fashion. In addition, behaviors may be connected via edges, to constrain the sequence of their execution (see, for instance, [3]).

We utilize a behavior-based recognition representation which serves as the basis for representing the modeled behaviors of the observed robot. In choosing a representation for recognition, we are fortunately not constrained by a specific behavior-based control methodology—since the representation does not express executable controllers—but instead can focus on common features to these methodologies. As a result, the representation is generic, and be used to represent controllers of various forms.

We follow previous work in representations for monitoring [37], and represent the behavior-based controllers of an observed robot as a directed acyclic connected graph, where vertices denote behaviors, and edges can be of two types: vertical edges that decompose top behaviors into sub-behaviors, and sequential edges that specify the expected temporal order of execution. Temporal edges may form cycles, but decomposition edges may not: A behavior cannot be its own parent, but may be selected again after it has already been selected and terminated.

Each behavior has associated with it a set of conditions on observable features of the robot or world that specify the settings under which observations are said to match the behavior. The behavior may also have associated preconditions and termination conditions that may be observable, and may thus be used to include or exclude it from an hypothesis. For example, lets take the existing robotic soccer teams [34], the kick-to-goal behavior has the following preconditions: The ball must be visible, the distance to the ball is within a given range, and the opponent goal is visible within shooting distance. If all the above conditions are satisfied, the behavior is applicable.

A behavior graph which includes all possible behaviors that the observed agent may execute (its complete behavioral repertoire) is called the *behavior library*. Typically, a behavior library has a single dummy root node. At any given time, the observed robot is assumed to be controlled by a *behavior-path*, a root-to-leaf path of behaviors that follows decomposition edges. Figure 3.1 shows an example portion of a behavior graph, inspired by the behavior hierarchies of the robotic soccer teams (e.g., [34]). The figure shows decomposition edges (solid arrows) and sequential edges (dashed arrows). For presentation clarity, we show the decomposition edges only to the first (in temporal order) child behaviors.

Given a set of observations as to the state of the world and the agent within it, the behavior recognizer's task is to determine which of the behavior paths in the behavior library match the observations. For example, based on this behavior library (Figure 3.1), the behavior path $root \rightarrow defend \rightarrow turn \rightarrow with\ ball$ can be an hypothesis as to the current internal state of an observed robot. A set of such behavior paths would constitute a set of hypotheses.

An observed agent may change its internal state in two ways. First, it may follow the sequential edges, such that when no further sequential links are available, control goes back to the parent (which then continues using its own sequential edges, if they exist). Second, control may be interrupted at any time to respond reactively to the environment, and a new (first) behavior may be selected.

For instance, suppose a robot was executing $root \rightarrow defend \rightarrow turn \rightarrow with\ ball$, and then interrupted this behavior. It may now choose $root \rightarrow attack \rightarrow pass$, but not $root \rightarrow attack \rightarrow turn$. The figure does not show the observation conditions associated with behaviors. For instance, suppose we there is a feature

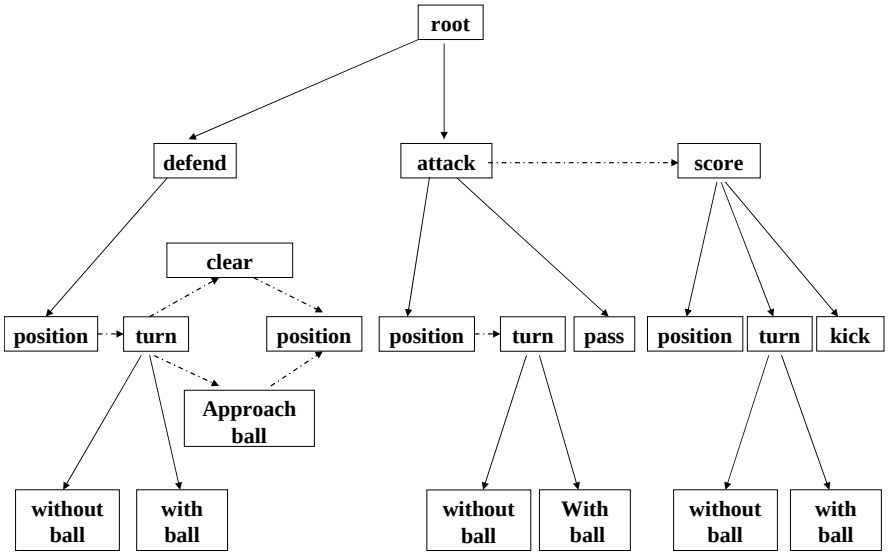

Figure 3.1: Example behavior graph.

$have\_ball$ whose value is true whenever the ball is observed to be in close proximity to the observed robotic soccer player. The behavior $kick$ may have a condition that specifies $have\_ball$ = $true$, while the behavior $approach\_ball$ would test for $have\_ball$ = $false$. Given appropriate observations, this behavior may be tagged as matching.

We assume that each behavior has self cycle, to allow for the duration of behavior execution. The behavior can be executed for several time-stamps. For example the duration of the approach ball behavior depends on the distance of the agent from the ball. If the robot is near the ball it can take one unit of time, and if the robot is far it can take 3 units. The same goes for behaviors in higher levels. For example, the score behavior can be executed several times, after the agent executed the attack behavior, the agent can execute several behaviors under the score behavior. For example, after executing $root \rightarrow attack \rightarrow pass$, the robot can choose the $root \rightarrow score \rightarrow position$, and afterward $root \rightarrow score \rightarrow kick$.

The next sections will address key algorithms in using such a behavior library to recognize the internal choices made by behavior-based agents, given multi-

feature observations of their interaction with the world. Chapter 4 presents an efficient method for matching observed features against behaviors, and tagging those that match. Chapter 5 presents algorithms that propagate such tags to make inferences of complete paths, in order to answer behavior recognition queries.

Chapter 4

Matching Observations to Behaviors

We begin by examining the first phase of behavior recognition, in which the observations made by the observing agent are matched against behaviors in the behavior library. The next chapter will address inference based on these matched behaviors.

In contrast with previous work, we consider complex observations, that consist of a tuple of observed features, including states of the world (e.g.,an observed soccer-playing robot's uniform number), actions taken by the robots (e.g., $kick$, $turn$), and execution conditions maintained by the robot (e.g., $speed$ = 200). It is likely, in realistic settings, that more than one behavior will match a set of observations, and this may result in multiple hypotheses as to the internal state of the observed robot.

Matching observations to behaviors can be expensive, if we go over all behaviors and for each behavior check all observed features. This, in fact, is what previous work essentially proposes. For instance, RESL goes over all behaviors, and for each, compares all observations against the expected observations given for these behaviors [19]. Since not all behaviors utilize all observed features in their associated observation conditions (see previous section), much of this effort may be wasted. Given the total number of features F, and the behavior graph of size L, RESL's worst-case matching run time will be $O(FL)$. In the best-case, where at most a single feature is associated with each behavior, its execution time will be $O(L)$.

To speed this process, we augment the behavior graph with a novel data-

structure, a *Feature Decision Tree* (FDT), which allows efficient mapping from observations to behaviors that may match them. An FDT works similarly to a machine-learning decision tree [27, 32], and is constructed similarly, but with important differences.

Each node in an FDT corresponds to an observation feature (e.g., velocity, heading, etc.). Each branch descending from a node, represents one of the possible values of this feature. Unlike traditional decision trees, each node also has pointers that point to the behaviors that test for the feature represented by the node. In this way, each node in the FDT divides a set of behaviors to subsets according to values of one feature. Thus determining the behaviors that match a set of observations features is efficiently achieved by traversing the FDT top-down, taking branches that correspond to the observed values of features, until a leaf node is reached. The behaviors a leaf points to are those that match the conjunctive set of observations.

An FDT can be built automatically. Unlike machine-learning decision trees, that are built based on examples of the target data, here we base the construction of the decision tree on the behavior graph which is given to us by the designer of the behavior recognition system. This behavior graph contains all the behaviors executable in principle by an observed robot. There is no uncertainty in determining which behaviors match a set of observations, and no need to prune nodes to prevent over-fitting ([27]). However, more than one behavior may match a set of observations. In case some behaviors do not test a feature, they are simply passed in the construction phase to all children FDT nodes, as they are consistent with all values of the features they do not test. Behaviors which have no associated observable features are excluded from this process, since they will appear in all nodes. Instead, these behaviors are handled in the propagating phase (section 5.1).

To generate the FDT, we first need to translate the behavior tree to set of instances, called training set. Each behavior in the behavior tree will represent one instance in the training set. An instance is a fixed set of values of features (e.g, velocity) and the class of the instance, in our case the class is the behavior itself. Note that each Behavior will appear just once in the training set, since the same behavior has the same features. In case that a behavior do not test a feature we put a question mark to denote missing values for that feature, otherwise we put the value of the feature. For example, let us consider three behaviors: $B1, B2, B3$

$classes \diagdown features$	a1	a2	a3
b1	T	?	T
b2	?	F	?
b3	T	T	T

Table 4.1: Example training set, generated from behaviors

and three boolean features: $a1, a2, a3$. Suppose the following conditions on the behaviors: $B1$ is possible if $a1 \wedge a3$, $B2$ if $\neg a2$, and $B3$ if $a1 \wedge a2 \wedge a3$. The resulting training set shown in table 4.1.

After generating the training set, the construction of the FDT is done similar to that of a decision tree with missing values [32]. Similarly to a decision tree, the construction of the FDT can use information gain to determine the most important features to test first, thus hopefully testing fewer features. We briefly review this well-known process here. The reader is referred to [27, 32] for details.

The FDT construction algorithm is presented below (Algorithm 1). First, we check if the instances can not be divided, meaning that a node points at only a single behavior, or there are no more features that can differentiate between the behaviors 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 behaviors that test it as uniformly as possible). We then create children FDT nodes for each of its values (lines 5–9), and recursively repeat the process of selecting a feature that best divides the behaviors associated with the node. The children constructed as follows: for each possible value of the selected feature, we select all instances that correspond to this value or have missing value. For each selected instance, we update its weight in the following manner: if there is a missing value, then we divide its weight in the number of the values of this feature, otherwise the weight remains the same. We also update the $testedFeatures$ set with the new tested feature. Then we recursively repeat on this process of selecting a feature that best divides the behaviors associated with the node with the new instances, new weights and the new tested features, and dividing accordingly.

To understand the selection of the best feature (line 3 of Algorithm 1 we need

Algorithm 1 formTree(Instances, weights, TestedFeatures)

if (there are no features to test) $\vee$ (single behavior) **then**
  return $createLeaf(Instances)$
$bestFeature \leftarrow$ best feature that was not tested
$createNode(bestFeature)$
for all possible values v of best feature $\neq$ missing value **do**
  $newInstances \leftarrow$ all instances with value v
  $newWeights \leftarrow$ calculate weights of $newInstances$
  $newTestedFeatures \leftarrow TestedFeatures \cup bestFeature$
  $formTree(newInstances, newWeights, newTestedFeatures)$

to understand the calculation of the information gain. There are some notations that accepted in the literature. we denote the training set with T. Suppose we have n possible values for the feature x, we divide the training set according to these values: $T_1, T_2, ..., T_n$. We denote the number of cases in T that belong to class C_i with $freq(C_i, T)$, and the total classes with k. We will also use the standard notation in which $\mid T \mid$ denotes the number of cases in set T. And we use F to denote the fraction of known values for feature x in the training set. Now, to compute the gain of feature x, we will use the three following equations as explained in [32]:

$$gain_x = F * (info(T) - info_x(T)) \tag{4.1}$$

$$info(T) = -\sum_{j=1}^{k} \frac{freq(Cj, T)}{\mid T \mid} \times \log_2(\frac{freq(Cj, T)}{\mid T \mid}) \tag{4.2}$$

$$info_x(T) = -\sum_{i=1}^{n} \frac{\mid T_i \mid}{\mid T \mid} \times info(T_i) \tag{4.3}$$

However, in our case $freq(C_i, T)$ is equal to one, since we have just one case for each class (each class is one behavior, that appear just once in the training set). And the number of elements in T, are the same as the number of classes in T, in our notations $k = \mid T \mid$ (because each class appear just once in T). So we can simplify equation number (2), and write it as follows:

$$info(T) = -\sum_{j=1}^{k} \frac{1}{\mid T \mid} \times \log_2(\frac{1}{\mid T \mid}) = \log_2(\frac{1}{\mid T \mid}) \quad (4.4)$$

Note that $info(T)$ and $info_x(T)$ are calculated only with the cases with known values for feature x. To illustrate this, we will go back to the previous example and compute the gain of the features a1,a2 and a3. Since $a1$ is a boolean feature, we will have two subsets of T: the first is T1, which contains all instances that the value of a1 is equal to one. And T2 will contain all instances that the value of a1 is equal to zero. So $T_1 = \{b1, b3\}$, and $T_2 = \emptyset$. The $info_{a1}(T)$ is as follows:

$$info_{a1}(T) = -\frac{2}{2} \times (\log_2(\frac{1}{2})) = 1 \quad (4.5)$$

$$info(T) = -\log_2(\frac{1}{4}) = 2 \quad (4.6)$$

So the gain is:

$$gain_{a1} = \frac{2}{3} \times (2 - 1) = 0.66 \quad (4.7)$$

To compute the gain of a2, we will do the same. T is divided into: $T_1 = \{b3\}$ and $T_2 = \{b2\}$.

Now, we compute $info_{a2}(T)$:

$$info_{a2}(T) = \frac{1}{2} \times (\log_2(1)) + \frac{1}{2} \times (\log_2(1)) = 0 \quad (4.8)$$

$$gain_{a2} = \frac{2}{3} \times (2 - 0) = 1.33 \quad (4.9)$$

The gain of $a3$ is equal to the gain of $a1$. So, the best attribute in this case will be $a2$. And the FDT of this example illustrated in figure 4.1. Note that just on the leafs there are pointers in the FDT, and not as in figure 4.1.

Figure 4.2, shows the connection between the FDT and the behavior graph. It

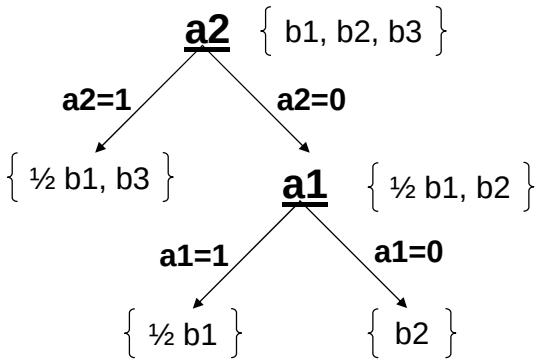

Figure 4.1: simple example for FDT

shows a portion of an FDT using features associated with behaviors in Figure 3.1. Each behavior executed by the robot, can be identified according to observed features, such as: Distance from other players, $have\_ball$, opponent goal visibility, uniform number of agent. The FDT separates the behaviors according to the values of these features. To determine matching behaviors, the matching algorithm first checks the $have\_ball$ feature. Based on its value, it continues the appropriate branch to test in sequence other features, until it finally reaches a leaf node. This leaf node will have pointers to all instances of the behaviors associated with it in the behavior graph. For instance the leaf-node for $position$ will have four separate pointers into the behavior graph in Figure 3.1. Note that since the behavior $turn$ is applicable regardless of whether $have\_ball$ is true or false, a node associated with it will appear in both left and right subtrees of the $have\_ball$ root node.

The Matching algorithm (Algorithm 2) matches the observations to the behaviors in the behavior tree using a FDT. The Match algorithm operates as follow: when observation is made about an agent we traverse on the FDT according to the values of the observed features until we get to a leaf (lines 2–4). Then, after getting to the appropriate node in the FDT, we have pointers to the relevant behaviors in the behavior tree. So, we return these pointers that match this node (line 5). Thus every feature is tested at most once (according to how we built the FDT), and from this go to the relevant behaviors in the behavior tree.

Matching behaviors to observations is efficiently done using an FDT, by fol-

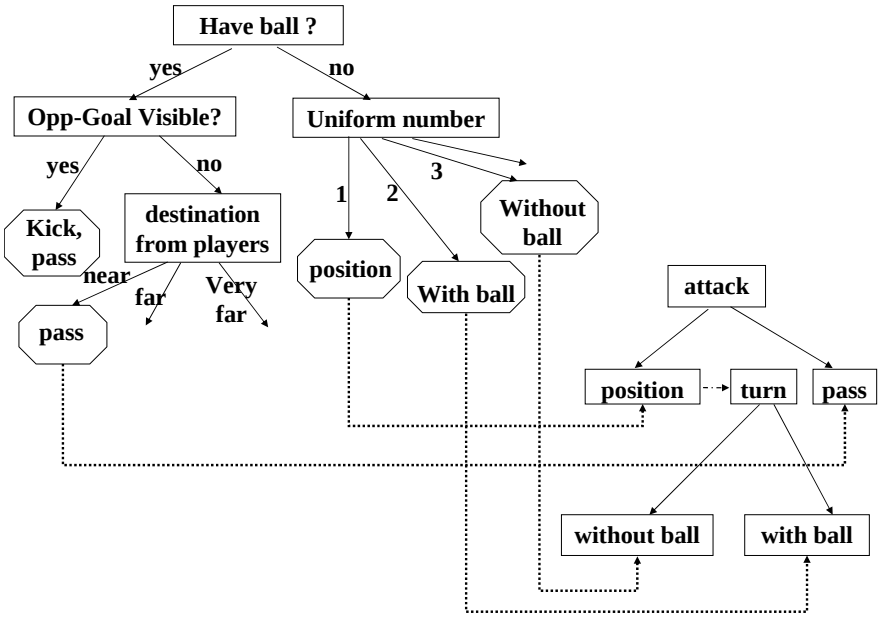

Figure 4.2: Example FDT with behavior tree.

Algorithm 2 Matching(Observations F, Timestamp t, Behavior graph bG, Feature Tree fdt)

1: $v \leftarrow root(fdt)$
2: **while** v is not a leaf **do**
3: $\quad i \leftarrow featureIndex(fdt, v)$
4: $\quad v \leftarrow child(fdt, v, F[i])$
5: return all behaviors in bG that match behavior in v

lowing a root-to-leaf path along the height of the tree. The height of an FDT is (in the worst case) $O(F)$ where F is the number of the features. This would be true only if a behavior test all available features, an unrealistic case. We believe that in realistic settings, the height of the tree would be closer to the best case of $O(logF)$. This result should be contrasted with previous algorithms such as RESC or RESL. In these algorithms, the matching step goes over all behaviors, and for each behavior tests all features, therefore the time complexity of these two algorithm for the matching step is $O(FL)$, where L is the number of vertices in the behavior graph.

The number of behaviors pointed to by an FDT leaf is $O(L)$ in the worst-case, where L is the number of nodes in the behavior graph. $O(L)$ is an unrealistic worst case, since it implies a case where a set of observations matches all behaviors in the library—thus recognition is futile.

Overall, the run time complexity of the process in the worst case is $O(F + L)$, and $O(logF)$ in the best-case.

The output of the matching phase is all behaviors that do not contradict the observations. Therefore, behaviors that were not included in the output, are definitely not matching (except those with empty features). The matching behaviors are given in Breadth-first search (BFS) order. i.e., parents will appear before their children. This fact does not influence on the matching process, but helps in the propagating process (discussed in the next chapter)

Chapter 5

Queries

We now turn to presenting symbolic behavior recognition (SBR) algorithms, that utilize the representation above. These algorithms answer two types of queries: (i) What are the possible current states of the observed robot (Section 5.1); and (ii) What are the possible sequences of states of robot, given the observation history (Section 5.2).

5.1 Current State Query

This query answer the question: what are the possible paths in the behavior graph, that the robot is currently executes? or in other words what are the hypotheses regarding the current state of the robot. The answer that the algorithms give is complete, but not accurate, i.e, the answer will include the correct hypothesis, but can include other hypotheses beside the correct one (hypotheses that based on given observations, could not been disqualified by the algorithm). The algorithms we present here are more accurate and more efficient than previous algorithms (e.g., RESL [19]). Unlike previous symbolic algorithms, SBR algorithms take into account previous observations in addition to the current observation, without any additional space or runtime overhead. The algorithms execute in two phases: (i) matching phase; (ii) tagging and propagating phase. The matching phase has been described previously. The propagation phase is described below.

Once matching behaviors are found, they are tagged by the time-stamp of

the observation. These tags are then propagated up the behavior graphs, so that complete behavior-paths (root to leaf) are tagged to indicate they constitute hypotheses as to the internal state of the observed robot when the observations were made. However, the propagation is not a simple matter of following child and parent edges.

One complication in tagging is that a behavior may match the observations (and is therefore tagged), and yet it cannot be a part of a valid hypothesis, when a history of observations is considered, i.e., it is *temporally inconsistent*. For instance, suppose that the first set of observations match the $turn$ behavior (Figure 5.1). The FDT would point the propagation algorithm to the three instances of $turn$, under $defend$, $attack$, and $score$. However, only the behavior instance under $score$ is valid, since it is the only instance in which $turn$ could have been selected without first going through $position$. Since this is the first set of observations, and assuming no observations were lost (an assumption we address in Section 8), it is impossible for the two other $turn$ instances to be valid, since they strictly follow a $position$ behavior, which was not previously matched. This reasoning about hypothesis consistency over time is a key novelty compared to previous symbolic behavior recognition algorithms (e.g., RESC and RESL [38]).

Another complication in tagging is that a behavior may match the observations, but its parent will not match the observation, Or in the opposite way, the behavior will match, but none of the children under this behavior will match .In both cases the behavior should be disqualified, since the matching phase should have returned all possible matching behaviors. For example, suppose, we got from the matching phase the following behaviors: $attack$, $withoutball$ and $position$ (Figure 5.1). The $withoutball$ behavior should be disqualified, since the features of the $turn$ behavior failed to match the observation. The only path that should be tagged is this stage: $root \rightarrow attack \rightarrow position$.

The propagating process is formalized in the Propagate algorithm (Algorithm 3). The propagate algorithm contains two parts: (i) *Propagate up*—tagging and propagating up time-stamps of the matching behaviors, according to time-stamp constrains (Algorithm 4); (ii) *Eliminate*—eliminating disqualified behaviors, i.e., erase tags from behaviors that were tagged on the $PropagateUp$ process, but none of their children were tagged. The propagate algorithm operates as follows: Lines

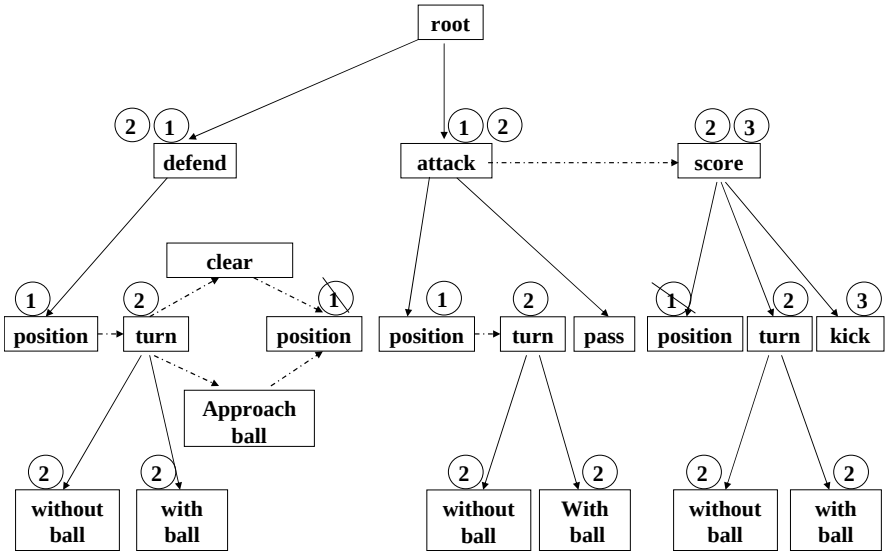

Figure 5.1: Example of the propagating process.

1–3 for each matching behavior it calls to the $PropagatingUp$ algorithm (Algorithm 4). If the $PropagateUp$ did not succeed in propagating up the time-stamp, then all tagged behaviors in this iteration will be erased (Lines 4–5). Otherwise, we will move on to the next matching behavior. After going over on all matching results, we will start the $Eliminate$ process(Lines 6), which erase tags from all disqualified behaviors, i.e., behaviors that match but their children do not match (5).

Algorithm 3 Propagate(Matching Results $matchRes$, Behavior Graph g, Time-stamp t)

1: **for all** $v \in matchRes$ **do**
2: $Tagged \leftarrow \emptyset$
3: **if** $\neg propagateUp(v, t, Tagged)$ **then**
4: **for all** $a \in Tagged$ **do**
5: $delete\_tag(a, t)$
6: $Eliminate(matchRes)$

The *Propagate up* algorithm (Algorithm 4), which is called for each of the behaviors that match the observations, takes a pointer to a matching behavior, and

tags behaviors using time-stamps to keep track of the order in which the hypotheses are formed. It exploits the sequential edges and the time-stamps to disqualify hypotheses that are inconsistent given *a history* of observations. It also disqualifies behaviors that were matched, but their parents were not tagged. To do so, it relies on the fact that the matching phase returns the matching behaviors in BFS order, therefore in the propagating up phase, first the father will be tagged and afterward, its children. This fact enable us to disqualify inappropriate behaviors.

The $propagateUp$ algorithm operates as follow: Lines 4–13 climb up the graph, tagging the behavior-path towards the root. Propagation is determined using the conditions (lines 5–6, Algorithm 4. These conditions are the key to the temporal validity of the hypothesis. First, the propagate algorithm check the parent validity: parent is tagged with time-stamp t or contains empty features (Line 5). Second, it checks the node in question, there are three cases: (a) the node in question tagged at time $t - 1$; or (b) the node follows a sequential edge from a behavior 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 (for instance, if another behavior 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 caused in climbing up the graph (line 6 3).

The *Eliminate* algorithm (Algorithm 5), is called after we tagged with current time-stamp all matching behaviors, and propagated up these tags. In this stage, we erase tags from behaviors that are tagged, but none of their children were tagged. The algorithm operates as follows: line 1 go over on all matching behaviors, and for each matching behavior checks if the behavior tagged and if exists at least one child that is tagged with current time-stamp (line 2). If all children were not tagged with current time-stamp, it erase the current time-stamp from the questioned behavior(line 3), and goes up to check the parent (line 4).

Figure 5.1 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 $position$ behavior. At time $t = 1$, Propagate (Algorithm 3) 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

Algorithm 4 PropagateUp(Node v, Behavior Graph g, Time-stamp t)

```
1: T ← ∅
2: propagateUpSuccess ← true
3: v ← w
4: while v ≠ root(g) ∧ propagateUpSuccess ∧ ¬tagged(v, t) do
5:    if tagged(parent(v), t) ∨ features(parent(v)) = ∅ then
6:       if tagged(v, t − 1) ∨ ∃PreviousSeqEdgeTaggedWith(v, t − 1) ∨
         NoSeqEdges(v) then
7:          tag(v, t)
8:          Tagged ← tagged ∪ {v}
9:          v ← parent(v)
10:         propagateUpSuccess ← true
11:      else
12:         propagateUpSuccess ← false
13:   else
14:      propagateUpSuccess ← false
15: return(propagateUpSuccess)
```

Algorithm 5 Eliminate(Matching Results $matchRes$, Behavior Graph g, Time-stamp t)

```
1: for all v ∈ matchRes do
2:    while tagged(v, t) ∧ ¬∃ChildTagged(t) do
3:       delete_tag(v, t)
4:       v ← parent(v)
```

$position$ instance under $score$ is initially tagged, but in propagating the tag up, 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 $root \rightarrow attack \rightarrow position$.

At time $t = 2$, suppose the observations match the $turn$ behavior (three instances). The tag $t = 2$ propagates successfully up and down in the behavior tree, for all instances. Now, there are six possible hypotheses (we omit the common $root$ prefix): $defend \rightarrow turn \rightarrow without\ ball$, $defend \rightarrow turn \rightarrow with\ ball$, $attack \rightarrow turn \rightarrow without\ ball$, $attack \rightarrow turn \rightarrow with\ ball$, $score \rightarrow turn \rightarrow without\ ball$, $score \rightarrow turn \rightarrow without\ ball$. Now, we can not decide which of the three main behaviors took place: $defend, attack$ or $score$. However, getting the next observation can disambiguate the hypotheses. If we will next observe a $clear$ or $approach\ ball$, then it would be clear that the observed robot is executing the $defend$ hypothesis. Otherwise, we can eliminate this hypothesis. In other words, we can exploit negative evidence to disambiguate the hypotheses space. This process is tightly coupled to the hypothesis generation phase, described next.
Complexity Analysis. For each behavior instance that matches the observations, the entire propagation traverses the height of the behavior graph, and may thus take $O(L)$ in a theoretical worst case in which the behaviors form a degenerate hierarchy. Realistically, we believe the height of the graph tree will often be closer to $O(logL)$.

This complexity is the same for previous algorithms (e.g., RESC [38]), despite the fact that they do not consider a history of observations, admit temporally-inconsistent hypotheses, and cannot answer queries as to hypothesized sequence of states.

5.2 History of States Query

This query answer the question: what were all possible sequences of behaviors that the robot executed from time $t = 0$ until the current time $t + k, k > 0$? Like the answer of the *current state query*, the SBR response to this query is complete, but not accurate. However, the answer become more accurate than in *current state*

query , as we exploit *negative evidence* to disambiguate the hypotheses space. Previous algorithms (e.g., [10, 19]) can not answer this type of queries. Here we present an algorithm, that answer this type of queries in efficient way.

We begin by defining *negative evidence* [10], as an observation at time t, whose occurrence at this time contradicts hypotheses based on observations made up until time $t - k, k > 0$. For example, let us take behaviors A, B, C, D with temporal constrains: $A \rightarrow B \rightarrow C$ or $A \rightarrow B \rightarrow D$. If we observed A at $t = 0$, $B, t = 1$, and $D, t = 2$, then D serves as negative evidence with respect to hypothesis $A \rightarrow B \rightarrow C$. It is the *lack of observation* of C that rules out the hypothesis.

Generating hypotheses about the current selected state (behavior path) is described in the previous section: Given the latest time t_0, we traverse the behavior-graph, identifying complete behavior paths that are tagged $t = t_0$. The set of these behavior paths constitutes the response to this type of query.

However, generating hypotheses as to the *sequence* of states that was selected over time is not a simple matter of enumerating combinations of the above queries for times $t = 0, t = 1 \ldots, t = t_0$. The reason for this is that new hypotheses, generated at some time t_0, may serve to rule out hypotheses that successfully matched at time $t < t_0$, by exploiting failures to observe expected behaviors, i.e., negative evidence as defined above.

Before discussing the hypotheses generation method, it would be useful to see an example of how reasoning about a sequence of behavior paths can lead to using negative evidence. Suppose that after having made the observations at times $t = 1$ and $t = 2$ in the example of the previous section, we now make observations at time $t = 3$ that match $kick$. The $score$ behavior is the only behavior consistent with $t = 3$, though both $defend$ and $attack$ are tagged for times $t = [1, 2]$. However, after having made the observation at $t = 3$, we can safely rule out the possibility that $defend$ was ever selected by the robot, because $score$ can only follow $attack$, and the lack of evidence for either $clear$ or $approach\ ball$ at time $t = 3$ (which would have made $defend$ a possibility at this time) can be used to rule it out. Thus we infer that the sequence of behavior paths that was selected by the robot is $attack \rightarrow position$ (at $t = 1$), $attack \rightarrow turn$ at $t = 2$ (though we cannot be sure which one of $turn$'s children was selected), and finally

$score \rightarrow kick$.

We now turn to the hypothesis generation method. Extracting all paths is not trivial (since as we saw, some successful tags at time $t < t_0$ are invalid at $t = t_0$. Here we present an incrementally-maintained structure that holds hypotheses according to time stamps. One advantage is that with every time stamp t, we can use the structure to eliminate hypotheses that were tagged at time $t - 1$, that have become invalid. Another advantage is that the algorithm is flexible, it is not necessary to call it after each matching-propagating steps, but according to queries.

We use a connected graph G', called *Hypotheses Graph*, whose vertices correspond to successfully-tagged behavior paths in the behavior graph (i.e., hypotheses). Edges in G' connect hypothesis vertices tagged with time stamp t to hypothesis vertices tagged with time stamp $t + 1$. G' is therefore built in levels, where each level represents hypotheses that hold in each time stamp. For each set of observations made at time t_0, we add to G' a level t_0 all possible hypotheses that were tagged $t = t_0$ and propagated successfully in the behavior graph. We then create edges between vertices $x_1, \ldots, x_n$ in level t to vertices $y_1, \ldots, y_m$ in level $t - 1$ in the following manner: If x_i is not part of a sequence (i.e., it is a first child), then we connect x_i to each vertex y_j (j=1...m); otherwise, if x_i is part of a sequence, we connect x_i to y_j (j=1...m) if any of the behaviors in y_j has a sequential edge to any behavior in x_i. If x_i is equal to y_j, we connect them, since we assume that we have durations.

The hypotheses graph is built based on the propagated time-stamps in the behavior library (Section 5.1). This fact, make our approach different and more efficient, from other graph-based approaches in behavior recognition (for example [14]). First, the hypotheses graph utilizes the negative evidence property. Second, it can be built in any stage, according to the query i.e, it can be built after each observation at time t, or as needed, since all the relevant data is saved in the behavior library's time-stamps. Third, not all the levels need to be built, we can generate the graph in time t, with k previous levels (it may be less accurate, but still complete). The only algorithm of which we are aware that utilizes negative evidence do not have these capabilities.

To generate all sequences of behavior paths that are consistent with the obser-

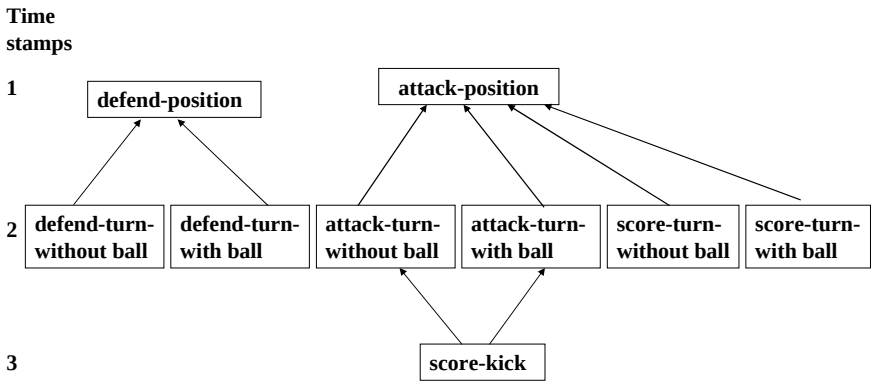

Figure 5.2: An example extracting graph G'.

vations, we traverse G' from level to level, keeping track of all G' paths that take us from the last level (the most recent observation) to the first level. This can be done incrementally, after each observation is made, or it can be made only when needed.

For example, based on the behavior tree in Figure 3.1, we construct G' (Figure 5.2). In the first level, we put all paths in the behavior tree that were tagged with time-stamp 1, in the next level we put all paths that were tagged with time-stamp 2. Now, from each node in time-stamp 2, we check which nodes can be appropriate in time-stamp 1. The $t = 1$ node $defend \rightarrow position$ can be connected to the $t = 2$ nodes $defend \rightarrow turn \rightarrow without\ ball$ and $defend \rightarrow turn \rightarrow with\ ball$, because there exist sequential edges in the behavior graphs that connect $position$ to $turn$ under $defend$. Similarly, $attack \rightarrow position$ has edges to $score \rightarrow turn \rightarrow without\ ball$ and $score \rightarrow turn \rightarrow with\ ball$. Once we add the observations for $t = 3$, the various $defend$ hypotheses have no link to time $t = 3$. If we now go back to asking what hypotheses exist for the current behavior paths at time $t = 2$, we will get $attack \rightarrow turn \rightarrow without\ ball$ and $attack \rightarrow turn \rightarrow with\ ball$. Here we can see the advantage of using G': Out of six hypotheses that matched the observations until time-stamp 2, four hypotheses are eliminated once we incorporate the evidence in time-stamp 3.

Complexity Analysis. The worst-case runtime complexity of constructing G'

over N observation time steps is $O(NL^2)$, where and L is the worst-case number of behaviors that the matching algorithm had returned given a single observation. For each node in G' with time stamp t (of which there could be at most $O(L)$), we check all nodes in time stamp $t-1$ (again, $O(L)$), thus a factor of L^2 for each step of adding another level. However, Note that the L component is a purely theoretical worst-case, as it corresponds to a recognition system that simply returns all behaviors in the behavior graph.

Chapter 6

Lossy Observations

Real-world applications sometimes violate assumptions that are made in recognition systems. One common violation of assumption involve intermittent observation failures. This section addresses this challenge.

In Chapter 4, we showed how to efficiently determine which behaviors match a set of observations. An implicit assumption was made (present also in most related work) that all relevant features were in fact observables. However, in realistic settings, some features may be intermittently unobservable, e.g., due to hardware failures, communication errors, etc. Observations that are lost would fail the conditions associated with behavior, and thus the matching phase will fail.

We propose to use an augmented FDT, called LFDT (Lossy Feature Decision Tree), which has all the properties of FDT, but deals with lossy observations (figure 6.1). The LFDT representation is the same as FDT, except that for each node, we add an extra branch that represents a *missing value*. During construction of the LFDT, all behaviors 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 construct the LFDT, we need a preprocessing phase, which generate training set according to the given behaviors in the behavior graph. This preprocessing is exactly the same as done for the FDT, and was demonstrated in section 4. The

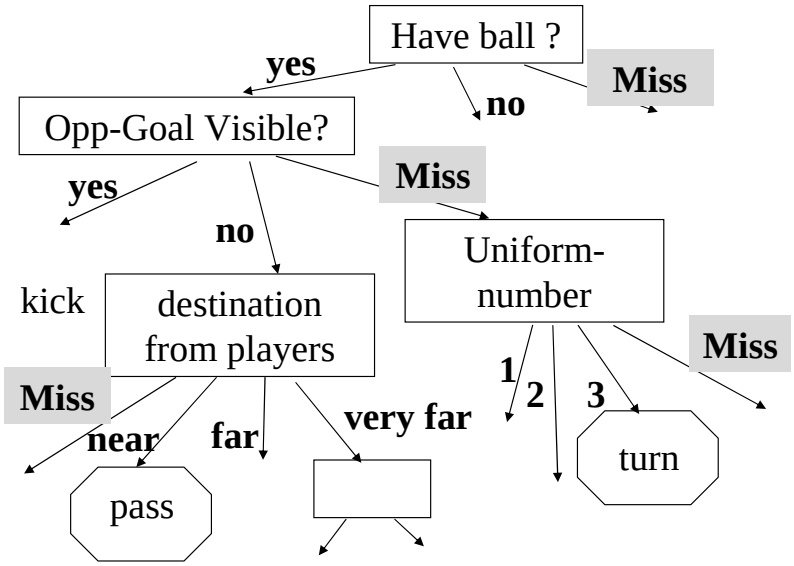

Figure 6.1: An example Lossy Feature decision Tree (LFDT).

construction of the LFDT, is also done similarly to the FDT construction.

The LFDT construction algorithm is presented below (Algorithm 6). This algorithm is a little bit different from the algorithm of the FDT (Algorithm 6). The change is in lines 5–8. The different is caused by the fact that here we add an extra branch which denote *missing value*. For this branch the instances remain as its parent (line 7), and the weights also remain the same (line 8). This child is different from his parent , only by the best features that it can select (line 12). Although, this node did not really divided the instances according to the best feature, we refer it as the best feature was already tested. Note that the computation of the information gain also remain the same for the LFDT.

The Matching algorithm also similar to that presented for the FDT (Algorithm 2). The LFDT Match algorithm operates as follow: when observation is made about an agent we traverse on the LFDT according to the values of the observed features, if we do not have the observed feature value, or the value is unreasonable, we turn to the *missing value* branch. This process is done until we get to a leaf.

Algorithm 6 formTree(Instances, weights, TestedFeatures)

```
1: if (there are no features to test) ∨ (single behavior) then
2:    return createLeaf(Instances)
3: bestFeature ← 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 ← Instances
8:       newWeights ← Weights
9:    else
10:      newInstances ← all instances with value v
11:      newWeights ← calculate weights of newInstances
12:   newTestedFeatures ← TestedFeatures ∪ bestFeature
13:   formTree(newInstances, newWeights, newTestedFeatures)
```

Then, after getting to the appropriate node in the LFDT, we have pointers to the relevant behaviors in the behavior tree, the same as we have in the FDT. So, we return these pointers that match this node. Thus, every feature is still tested at most once, and from this go to the relevant behaviors in the behavior tree.

Complexity Analysis. The runtime complexity of LFDT is the same as FDT (Section 4), though the size of the LFDT would be greater: (a) it will have more branches than FDT (extra branch for each feature); (b) its height may be deeper than FDT (because of the need to handle missing features at the leaves). However, the complexity will be still $O(F)$. To lower the size of the LFDT we can add an extra branch just to lossy features, i.e., not all features, but the ones we know we can lose. The space complexity of the LFDT is obviously greater then of the FDT, since we have an extra branch. However, it is almost the same as having more values to each feature.

Chapter 7

Experiments

To empirically evaluate the performance of the algorithms, we conducted an extensive set of experiments, varying a number of parameters that affect their performance. In particular, the performance of algorithms depends very much on the structure and size of the behavior library, as well as the set of observations presented to the behavior-recognition system.

We first describe the experimental setup and the parameters defining the scope of the experiments (Section 7.1). We then turn to presenting the results of the algorithms presented above; first, the efficient matching (Section 7.2), and then generating and extracting hypotheses (Sections 7.3 and 7.4).

7.1 Experiment Set-Up

To systematically evaluate the performance of the algorithms given the variety of possible behavior libraries, we built a *Behavior Tree Generator* which generates behavior-libraries (based on given parameters—see below), and an *Observation Generator* which generates sequences of legal observations, given a behavior library. All algorithms were implemented in C++ and tested them on Pentium 4 processor with 1GB of RAM and 2.40GHz CPU, in Linux.

The *Behavior Tree Generator* generates random behavior libraries that conform to the following parameters:

Top Level Branching Factor. The branching factor of the root node (number of

children for the root node). This corresponds to the number of different independent high-level behaviors in the library, referred to as number of roots in [10].

Depth. The depth of the behavior library, from the root.

Branching Factor. The branching factor of all nodes (other than root).

Sequential edges type. Following [10], we vary the ordering constraints between nodes in the library, using different sequential edges between nodes. There are six types (Figure 7.1):

Totaly ordered. All children with the same parent node, are connected with single sequential edges, i,e., siblings form a single chain.

First. First node will have ordering constraint (sequential edge) to all other nodes under its parent.

Last. All nodes will have single ordering constraint to the last node.

Partial A. Each node will have random number of ordering constraint, the number of constraints will be between zero to number of brothers of this node. Cycles will be prevented at generation.

Partial B. Each node will have zero or one sequential edge to its brothers.

Unordered. No ordering constraints between nodes.

One exception is that top-level behaviors (children of the root node) are always unordered.

Number of possible features. Sets the number of observable features (overall).

Number of features in each node. Sets the complexity of observations associated with each node, i.e., the number of features associated with a single observations.

Duplication. The fraction of top-level behaviors that are duplicated in order to generate ambiguous paths. To make sure some differences between duplicated still exist, the last leaf behavior in duplicated top-level behaviors is

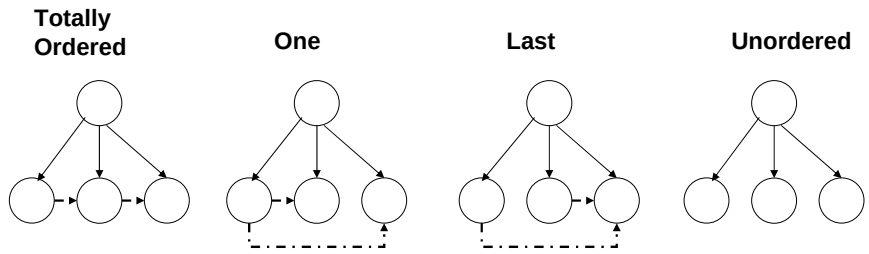

Figure 7.1: Sequential Links types. Partial A and Partial B are not shown, as they are non-deterministic, and may take different forms.

made different. For instance, a fraction of 0.4 means that 40% of the top-level behaviors are duplicates of others (i.e., approximately 80% of top-level behaviors are not unique—except for their last leaf).

The size of the behavior hierarchy is affected by the top-level branching factor, the normal-node branching factor, and by the depth. For the purposes of the experiments, we have fixed the normal-node branching factor at 3, and only varied the top-level branching factor and the depth.

Each behavior in the tree is uniquely identified, but may have the same set of associated features as other behaviors, in which case they will be considered equal—and will both match given the same set of observations. The inherent ambiguity of the behavior-hierarchy (how many different hypotheses are valid given a sequence of observations) is goverend by the duplication parameter (larger means more ambiguity), by the number of overall features (more overall features enable in principle greater differentiation between behaviors), and the number of features observable per behavior node (smaller number will cause less variety in behaviors, and thus increased ambiguity). In the experiments reported below, we chose values that are conservative for the techniques we developed, and thus represent worst-case scenarios: The number of overall features (which increases FDT size) was set at 10, creating the largest FDT to fit in the computer memory. The duplication fraction was set at 0.4. The number of features per behavior was set at 1, which essentially treats features as atomic, and thus lessens the expected effect of using the FDT.

The *Observation Generator* generates legal sequences of observations according to a given generated behavior library. To generate legal observation, it simulated execution and selection of behaviors. It randomly chooses a path in the behavior library and uses all the features in this path to generate observation. Then, according to the sequential edges, the Observation Generator chooses another path in the tree. The selection is done in the following manner: It first finishes all children under the parent, then if there is a sequential edge, it prefers to follow the sequential edge,otherwise it goes up to choose other child. Behavior will be selected again, until we visited all sequential edges from this behavior, and all children of the behavior.

7.2 Matching Observations To Behaviors

The first set of experiments tests the run-time of the observation matching phase, using an FDT (Section 7.2.1). It then explores the computational cost associated with FDT in terms of building run-time (Section 7.2.2).

7.2.1 Matching Runtime

We investigate how the run-time of the matching phase scales with the size of the behavior-hierarchy. The run-time using an FDT is contrasted with the matching run-time of the RESL algorithm [19], the most relevant of related works.

Three parameters affect the matching time: L, the size of the behavior library (which affects how many behaviors there are to match against), F, the number of overall features, and f, the number of features associated with each behaviors (of course, $f \leq F$).

As reported in Chapter 4, RESL's worst-case run-time in theory is $O(FL)$, and its best-case run-time is $O(fL)$. The FDT's worst-case run-time is $O(F + L)$, and its best-case is $O(log\ f)$.

In the following experiments, we varied L by varying the number of top-level behaviors (5,50,100) and the depth of the behavior library (3–5). F was fixed at 10, and f was varied between 1,3,5,7. For each of these values, we generated 180 random observations sets based on the given behavior-libraries, and averaged the

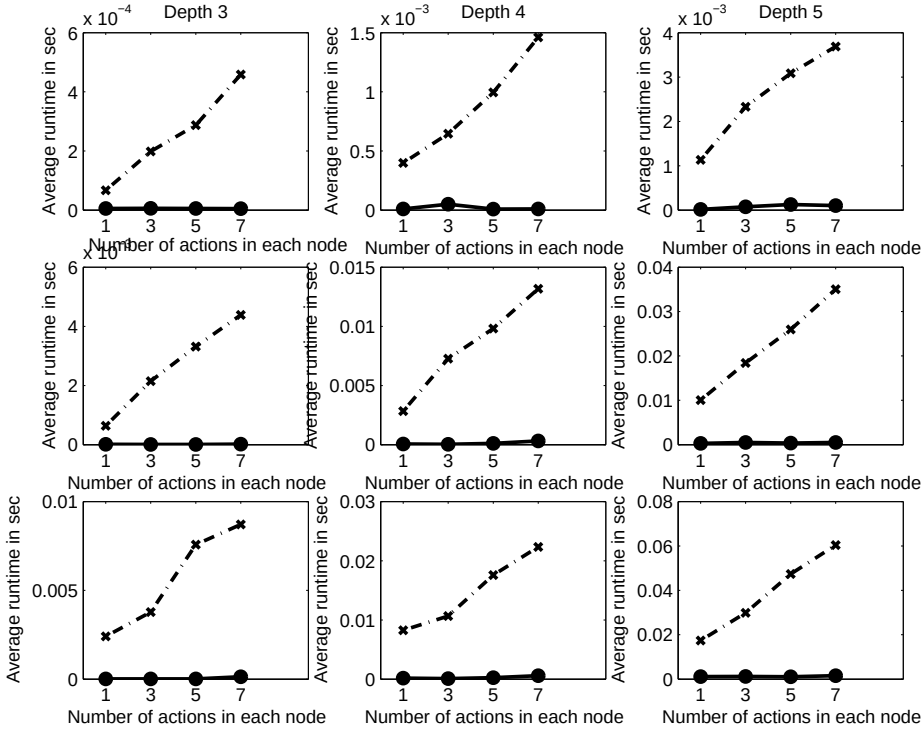

Figure 7.2: Average matching runtime of FDT and RESL, as a function of f for varying depth and top level roots

run-time for matching these using RESL and using a generated FDT.

The average runtime of the matching algorithms are shown in figure 7.2.1. The horizontal (X) axis shows the number of features associated with each behavior (f). The vertical axis shows the average matching time in seconds. First line is for 5 top-level behaviors, second line is for 50 top-level behaviors and third line is for 100 top-level behaviors. The columns represent varying depth of the behavior library (3–5). For example, top left figure the depth of the behavior library is fixed at 3, and the top-level behavior branching factor was fixed at 5. For each value of the number of features per behavior, we repeated the experiment 180 times.

Clearly, the use of the FDT leads to very significant improvements in the matching time, compared to RESL. Furthermore, its growth curve (with respect to the number of features associated with each behavior) indicates that its benefits scale up well as the observed behavior increases in complexity.

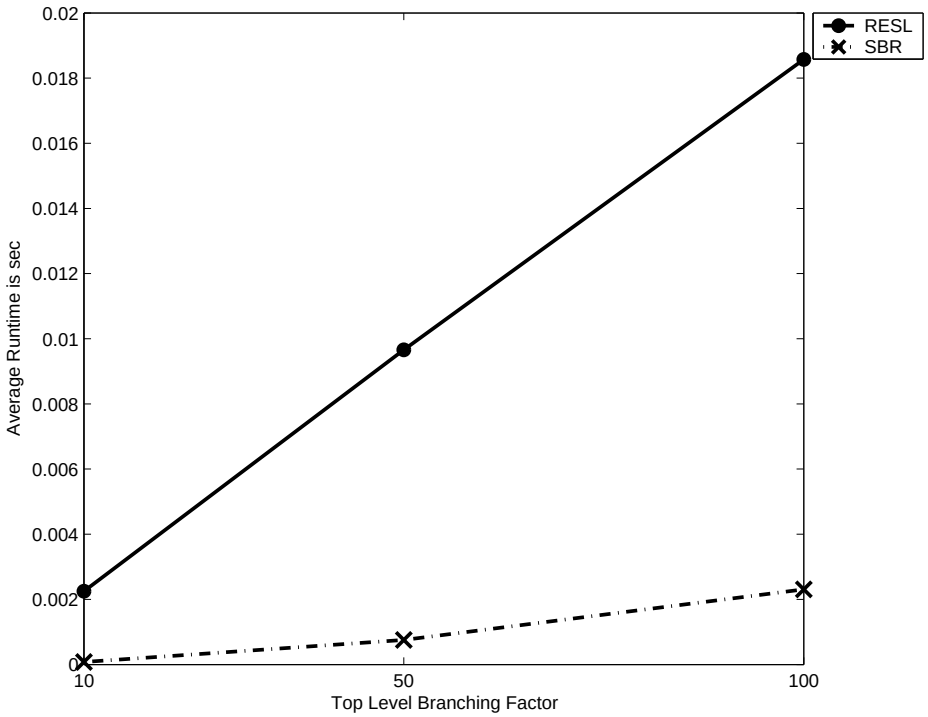

Figure 7.3: Average matching run-time of FDT and RESL, worst-case (for FDT), as library increases in size.

We now turn to the question of how this performance varies with the size of the behavior hierarchy. We focus on the worst-case scenario of the FDT in comparison with RESL, where the number of features per behavior is $f = 1$. We then varied L by varying the number of top-level behaviors (10,50,100) and the depth of the behavior library (3–6). Again, F was fixed at 10. For each of these values, we generated 720 random observations sets based on the given behavior-libraries, and averaged the run-time for matching these using RESL and using a generated FDT.

The results in figure 7.2.1 show that the FDT-based matching algorithm is significantly faster than the RESL algorithm, and the runtime gap grows with the size of the library. This result is expected, given the complexity analysis of RESL vs. the FDT-based matching.

Since, the action DB is fixed here to 10, naturally, the number of matching

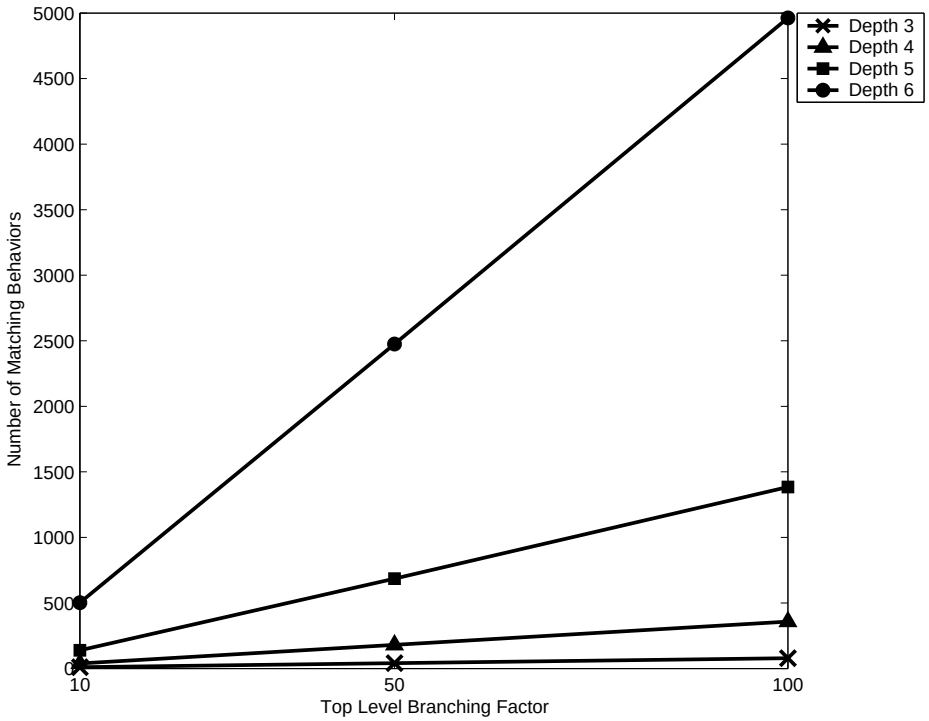

Figure 7.4: Matching Experiment: Number of matching behaviors

behaviors is also increase with the size of the tree (figure 7.4). This fact cause overhead that should be considered when exploring the matching runtime and the building *FDT* runtime.

7.2.2 FDT Build time

The use of the FDT leads to very significant savings in matching run-time. However, it does necessitates carry a one-time cost of building the FDT for usage. This section evaluates the this cost to argue that it is feasible.

We first explore how the FDT construction runtime changes as the number of features per behavior is varied. Figure 7.5 shows the average runtime for building the *FDT* as number of features per behavior varies between 1,3,5,7. The left figure is for top-level behaviors fixed to 50 and the right figure is for top-level behaviors fixed to 100. The depth of the behavior library (3–6). For each of these values, we generated 30 random observations sets based on the given behavior-libraries.

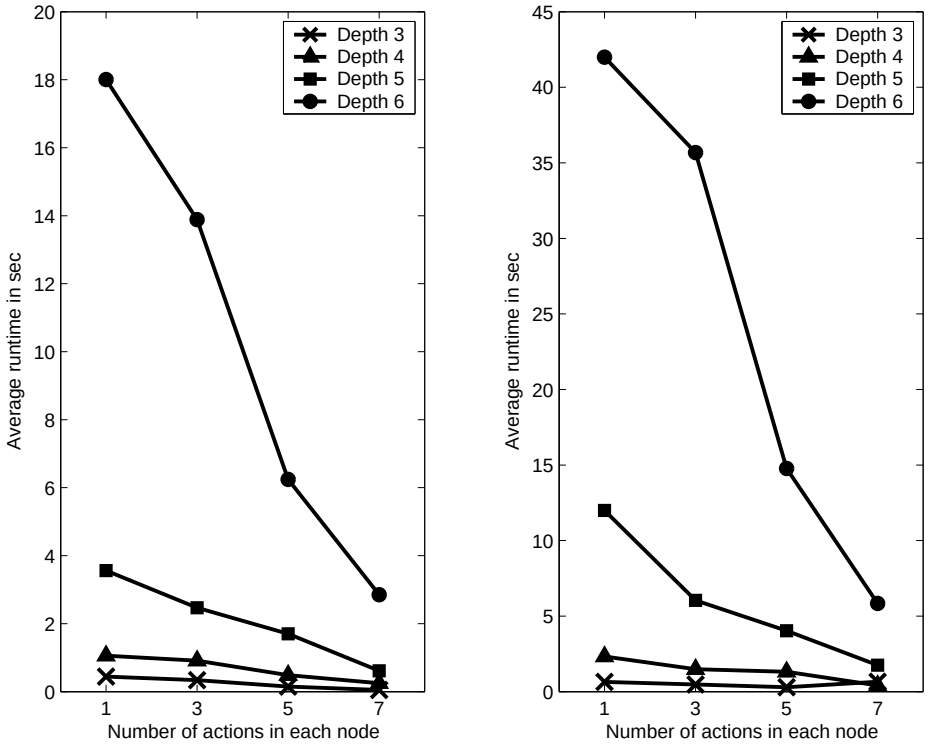

Figure 7.5: Matching Experiment2: Average runtime of building *FDT*

The average time for building the FDT decreases as the number of features in each behavior increases. The reason for this is that behaviors that do not test a feature simply pass down (in the construction phase) to all children FDT nodes, as they are consistent with all values of the features they do not test (as explained in Section 4).

We again focus on the worst-case scenario; here, the construction run-time when only a single feature is associated with each behavior. Figure 7.2.2 shows the average runtime for building the FDT for a behavior libraries of various sizes. The horizontal axis shows the depth of the library (3–6), while the different graphs correspond to the 10,50 and 100 top level behaviors. For each of these values, we generated 48 random observations sets based on the given behavior-libraries.The vertical (Y) axis shows the average run-time in seconds. While the graphs hint that the construction time may be exponential in the size of the hierarchy, we remind the reader that building the FDT is a one-time offline cost, while matching takes

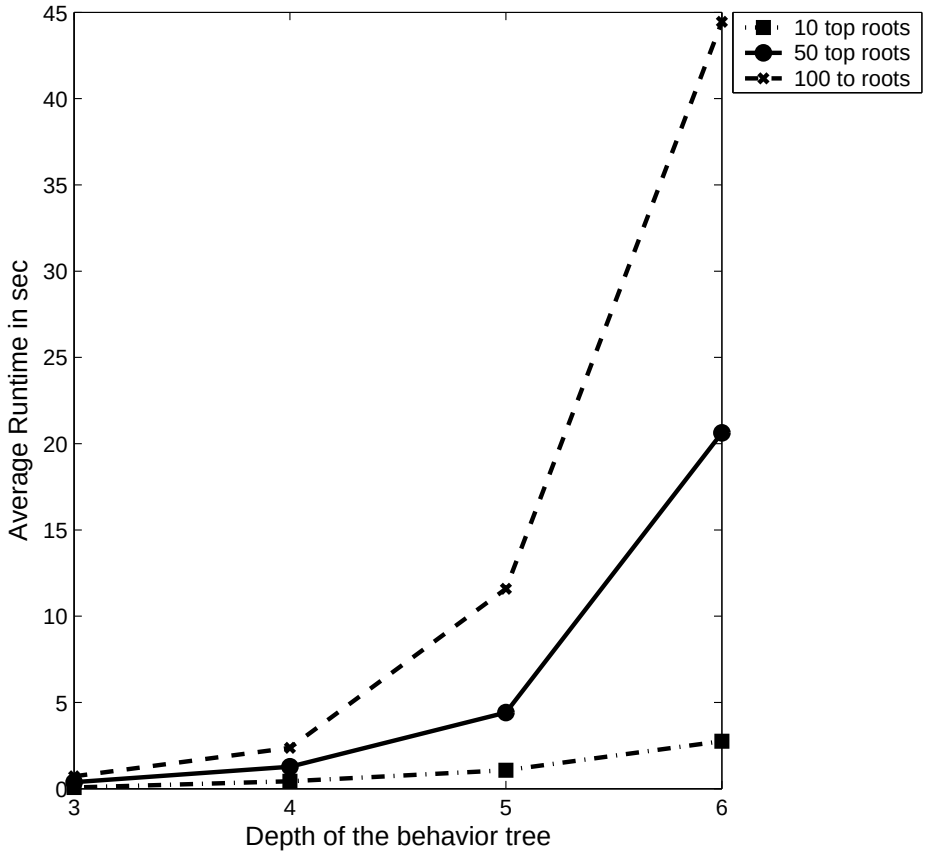

Figure 7.6: Average FDT construction runtime

place many time in realistic settings.

7.3 Current State Query

We now turn to evaluate our *SBR* algorithms with respect to their ability to answer the current-state query. We tested the SBR propagation algorithm (the key algorithm used in answering this query, once matching is done), in terms of scalability (efficiency) and accuracy. We contrast these results with the RESL algorithm [19].

	10 roots	50 roots	100 roots
Number Hypotheses SBR	5.47	12.39	20.93
Number Hypotheses RESL	9.75	29.17	53.37

Table 7.1: Average number of hypotheses after propagation, RESL vs. SBR.

7.3.1 Accuracy

A key advantage of SBR over RESL is its ability to use the history of observations, together with the sequential edges, to rule out certain hypotheses that match current observations, but are not feasible given the history of observations. Thus given a valid sequence of observations, and depending on the existence of sequential edges in the behavior library, we expect to see fewer hypotheses as to the current state of the observed robot, in comparison with RESL's output.

We fixed the number of features per behavior is $f = 1$ (since it does not affect directly on the propagating). We then varied L by varying the number of top-level behaviors (10,50,100) and the depth of the behavior library (3–6). Again, F was fixed at 10. For each of these values, we generated 720 random observations sets based on the given behavior-libraries.

Figure 7.3.1 compares the average number of hypotheses after propagating in SBR algorithm to the number of hypotheses after propagating in RESL algorithm.The results are also shown in table 7.1. More than 50% of the hypotheses we get in resl propagating algorithm, were ruled out by the SBR propagation algorithm.

The ability of SBR propagation to use the history of observations relies on the sequential edges to rule out hypotheses. Thus it makes sense to examine how different structures, in terms of sequential edges, affect the number of hypotheses generated by SBR.

Figure 7.8 shows the effect of various sequential edges types on the number of hypotheses. There are four graphs, each for different depth varying between 3–6. For SBR, the number of hypotheses depends on the type of sequential edges used in the behavior library. Totally-ordered behavior libraries allow SBR to maximally use past observations, and thus on average result in the fewest number of hypotheses. In contrast, unordered behavior libraries have no sequential edges,

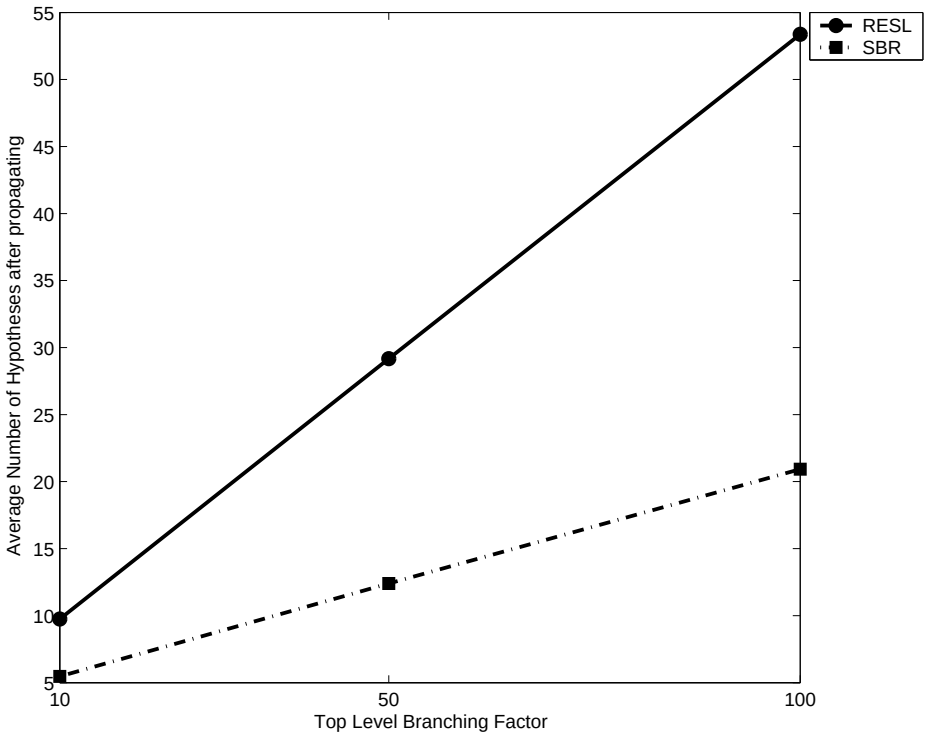

Figure 7.7: Average number of hypotheses after propagation, RESL vs. SBR.

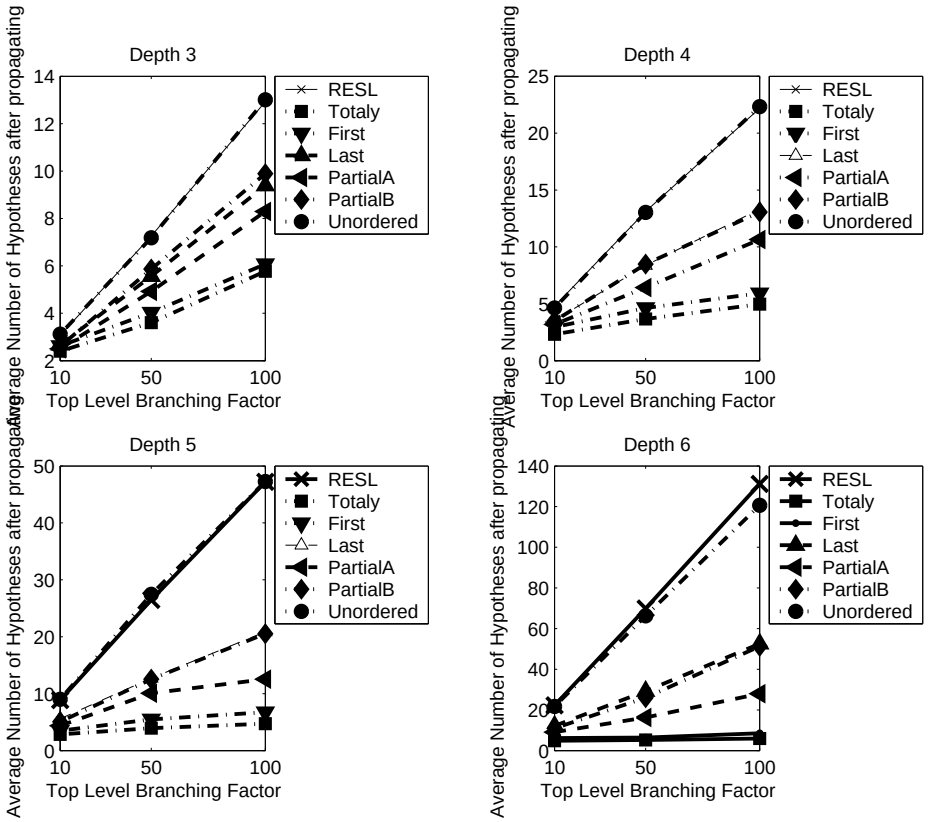

Figure 7.8: Average number of hypotheses after propagation, RESL vs. SBR, with different sequential-edge types.

and thus do not allow SBR to use a history of observations. Thus the number of hypotheses generated in this case is exactly the same as generated by RESL. The number of hypotheses in RESL algorithm is not affected by the type of sequential edges, so only a single solid line shows the average results of running RESL.

7.3.2 Runtime

Given the significant improvement in accuracy, one may expect that there is an associated significant computational cost to the use of the propagation algorithm. Surprisingly, this is not the case. Figure 7.9 shows the average run-time of the SBR propagation algorithm in the above experiments, in comparison to that of RESL. The horizontal axis shows the top-level branching factor, while the vertical

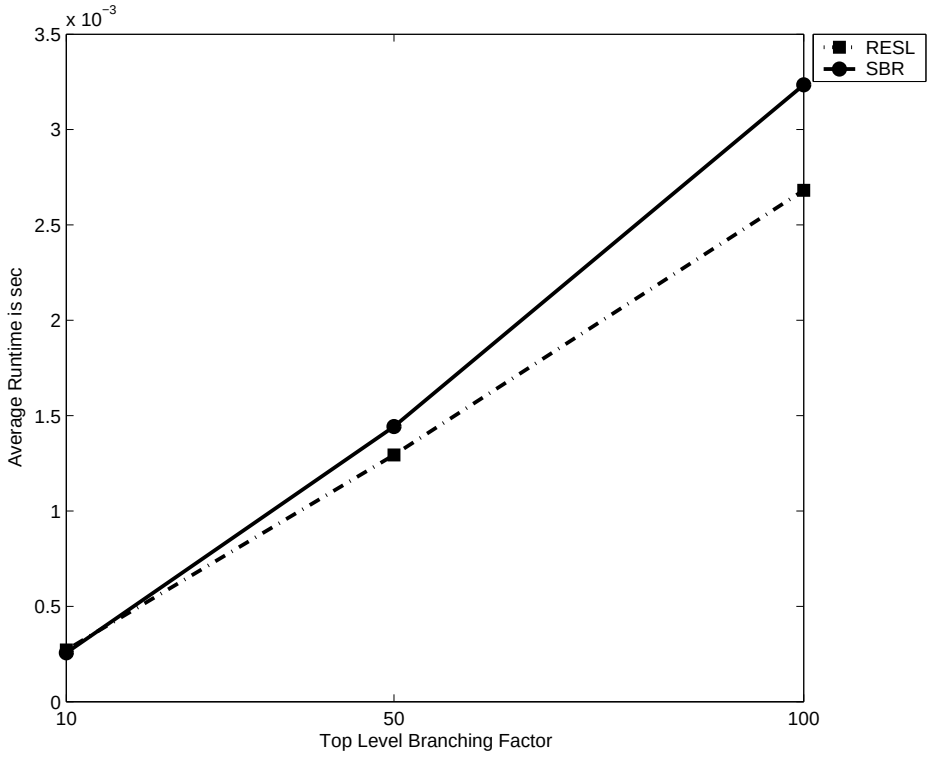

Figure 7.9: Average runtime of propagating Resl versus SBR

axis shows the runtime in seconds. RESL is only slightly faster than SBR. A close examination of the propagation algorithm shows that the only difference between the two propagation algorithms is that in comparison to RESL, SBR carries out a few additional checks (for incoming sequential edges) as its propagates time-stamps up and down the behavior hierarchy. Thus the addition to runtime is minor.

7.4 History of States Query

As mentioned in Chapter 5, there are three phases to answer this query: (i) matching; (ii) propagating; and (iii) generating hypotheses. The first two phases are done also to answer the *Current State Query*, and therefore were analyzed in section 7.3. In this section we will evaluate the third phase, generating hypotheses, which allows answering queries as to the possible sequence of behaviors se-

lected by the observed robot. This phase can be incremental to the previous two stages,i.e, done after each propagating phase, or after any number of matching-propagating iterations. Algorithms such as RESL do not have the ability to answer the *History of States Query*.

7.4.1 Generating behavior-history hypotheses

The hypotheses graph is used to disqualify hypotheses which are ruled out by negative evidence. Given an observation at time t, it is sometimes possible to rule out the suitability of an hypothesis as to the selected behavior of the robot at time $t-1$. Thus the hypotheses graph is has an important role in confirming and ruling out hypotheses as to current state at a given time.

However, the hypotheses graph is also used to generate hypotheses as to the sequence of behaviors selected by the observed robot over time (Section 5.2). Here, we want not only to determine the possible current state of the robot, but the possible sequence of states (leading to the current state). Given the sequential edges in a behavior library, it should be possible to rule out hypotheses over time (as we have seen), and thereby restrict the number of hypotheses as to the sequence of behaviors. This of course depends on the structure of the sequential edges in the behavior library.

Figures 7.10,7.11 and 7.12 show the number of possible behavior-history hypotheses evolving over time for 5,10,50 top-behaviors, for libraries with different types of sequential edges. In all graphs, the vertical X axis shows the observations from 1 to 10 (this is for observation sequences of length 10). The Y axis shows the number of hypotheses. The results are averaged over 30 trials, depths of 3–6, and over top-level branching factors of 5 7.10, 107.11, and 507.12. [Each point is thus the average of 30 trials.]

For example, the upper left figure of 7.11 notes trees with depth fixed to 3 with different types of sequential edges, after the first observation there are on average 2.4 possible hypotheses about the behaviors selected by the robots thus far. After the second observation, there is a slight increase in the average number of hypotheses, to 2.5 (i.e., there are—on average—between two and three possible sequences of behaviors that may have been selected by the robot leading to

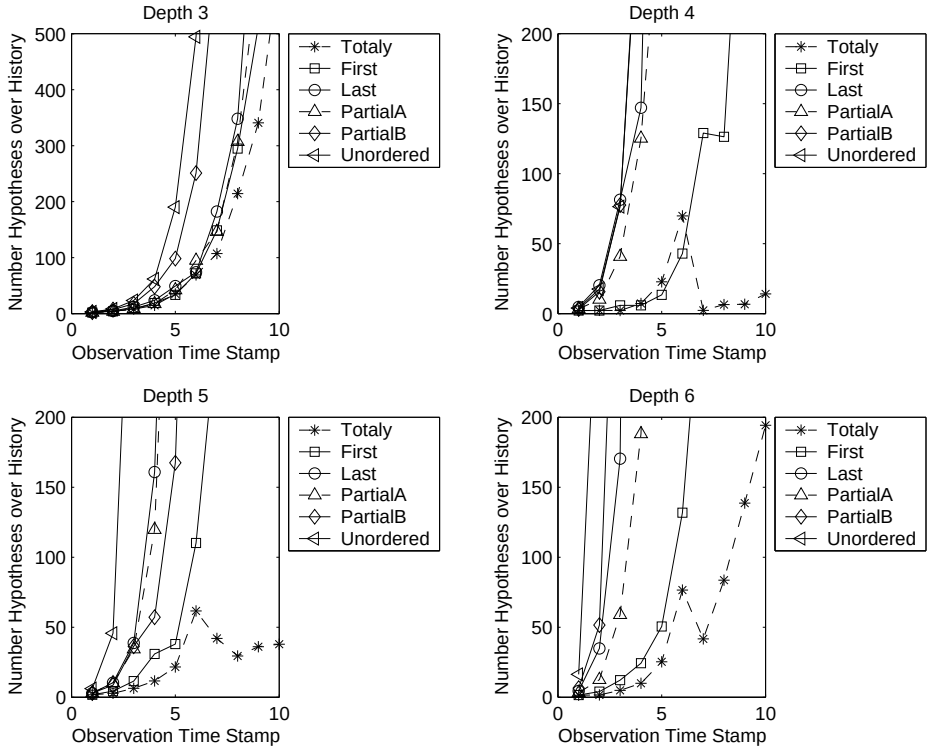

Figure 7.10: Number hypotheses over history for each sequential edge type, 5 top level behaviors

its current hypothesized state). After the third observation, ambiguity is reduced on average, as we have in average just 1.6 possible hypotheses. This is due to the characteristics of the artificial duplication factor, where the last leaf (the third, given the branching factor of 3) was artificially made different in otherwise duplicate top-level behaviors. After 10 observations, only 6.8 hypotheses are possible on average (the results are also displayed in Table 7.2).

In contrast, where there are no sequential edges (7.11), the ambiguity is very large since all possible combinations of all current-state hypotheses over time $t = 1 \ldots 10$ are possible. In other words, if we mark as H_t the number of current-state hypotheses at time t, the number of unordered state history hypotheses is $H_1 \times H_2 \times \ldots H_t$. Thus after 10 observations, for instance, the number of state history hypotheses in the unordered behavior library case is 13859 (compare to 6.8 for the totally ordered case).

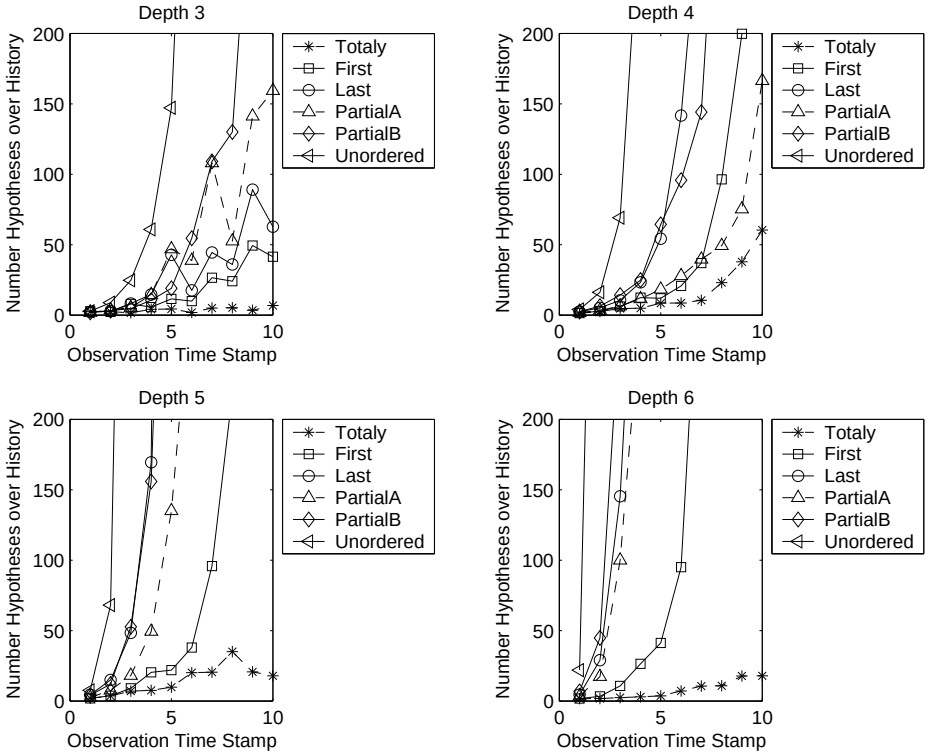

Figure 7.11: Number hypotheses over history for each sequential edge type, 10 top level behaviors

	Totaly	**First**	**Last**	**PartialA**	**PartialB**	**Unordered**
1	2.4	2.7	2.2	1.9	1.9333	2.7333
2	2.4667	2.3	2.7333	3.6333	3.2333	8.8333
3	1.6667	7.8667	8.2333	5.3333	4.5333	24.733
4	4.1	5.6333	14.633	14.467	10.567	60.867
5	4.3333	11.533	42.767	46.9	19.033	147.23
6	1.9	9.9	17.533	38.733	54.533	469.9
7	5.0333	26.5	44.467	108.03	109.2	1018.2
8	5.2333	24.033	36	52.5	130.07	2701.2
9	3.4	49.4	89.067	141.27	337.43	7805.3
10	6.8333	41.4	62.733	159.5	368.87	13859

Table 7.2: Number of Hypotheses over history

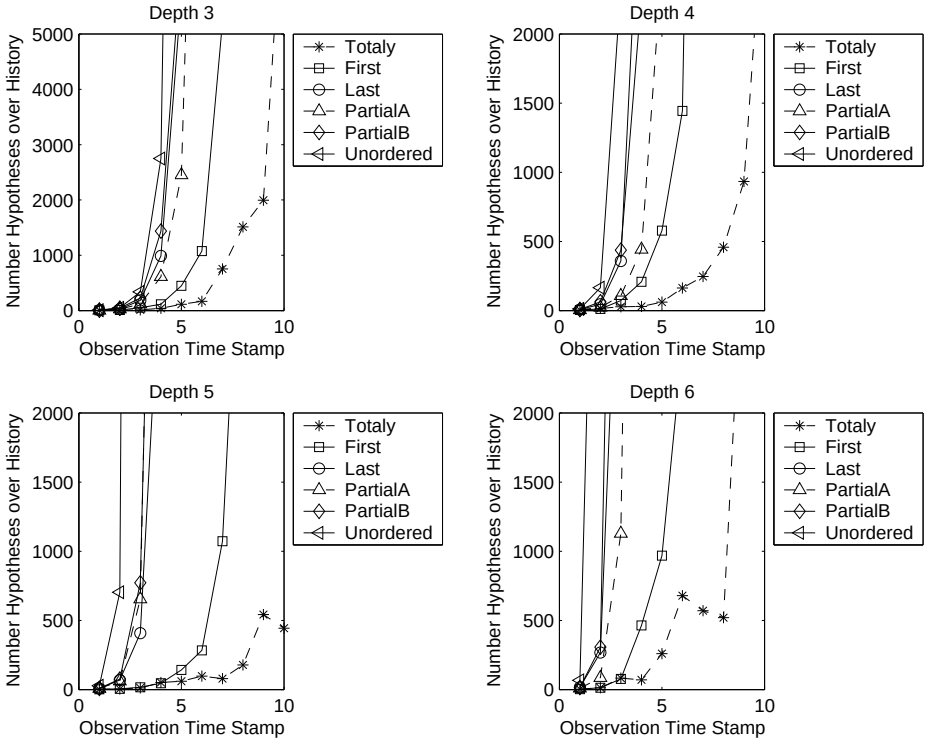

Figure 7.12: Number hypotheses over history for each sequential edge type, 50 top level behaviors

7.4.2 Runtime

The Generating Hypotheses phase is not expensive in means of runtime, compared to matching and propagating phase. Figure 7.13 compares the average runtime of the three phases mentioned above, for the following conditions: number of features per behavior is $f = 1$,L varying the number of top-level behaviors (10,50,100) and the depth of the behavior library (3–6). The average runtime shown Figure 7.13 represents the average runtime time to build the graph in each step and also to extract all paths from time stamp t=0, until current time stamp, where in each phase we use the results from the previous stage. Note that the extraction does not enumerate the resulting hypotheses—otherwise its runtime would grow combinatorically large—but simply marks them efficiently on the hypotheses graph.

An important feature of the SBR extraction phase is that it can take place at a different time than propagation. Indeed, if there's no need for answering state history query, the extracting phases is not needed at all (though the hypotheses graph may still be useful) to maintain direct pointers to current state hypotheses. Although we incrementally computed the hypotheses graph in the extraction phase, the average runtime results in Figure 7.13 would be the same if the process was instead carried out once, at the end of the sequence of observations.

Figure 7.14 differentiates the runtime involves in building and maintaining the hypotheses graph, in contrast with the runtime spent marking possible hypotheses from time $t = 0$ to the current time. The total runtime includes both of these components. We used the same conditions as above (number of features per behavior is $f = 1$,L varying the number of top-level behaviors (10,50,100) and the depth of the behavior library (3–6)). Each data point represents 2880 observations.

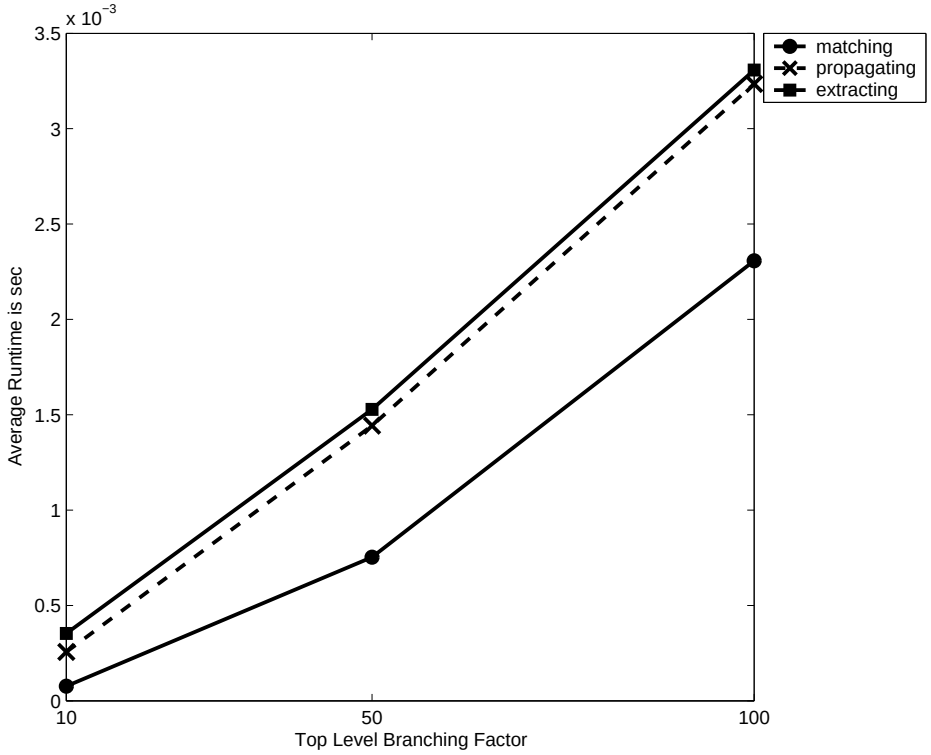

Figure 7.13: Matching, Propagating, and Extracting Average runtime

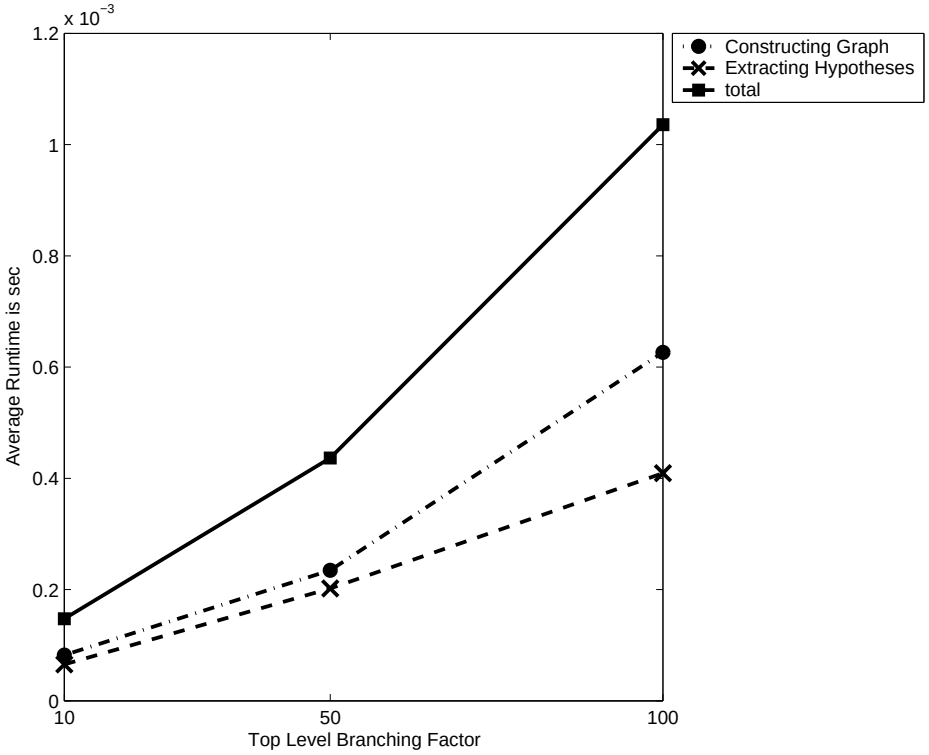

Figure 7.14: Extracting average Runtime

Chapter 8

Conclusion and Future Work

It is important for an agent to monitor other agents in order to carry out its tasks. To do this, agents must often rely on their observations of others, to infer their unobservable internal state, such as goals, plans, or selected behaviors. However, plan-recognition approaches to this task are insufficient for modern agents applications, such as robotics.

This thesis addresses this challenge by defining a behavior-based recognition representation and a comprehensive set of algorithms that can answer a variety of recognition queries. The algorithms we propose are efficient, and can handle important real-world challenges to existing techniques, such as intermittent failures in observations, behaviors with duration, etc. Surprisingly, we found that the bulk of these new features can be achieved in the same run-time complexity of previous algorithms—that lack these features.

In future work we intend to expand the algorithms we presented and test them on real world applications. Here we suggest some of the areas that should be explored:

Testing on dynamic, complex domains. In this thesis we explored and tested our algorithms on synthetic data that we created. To understand our contribution, we intend to test our algorithms on real world data taken from ModSaf domain, which is a commercially-developed virtual environment with synthetic helicopter pilot agents that carry out a variety of missions [35]. And on RoboCup, which is soccer simulation, with dynamic multi-agent

environment which requires real-time teamwork and coordination [36].

Multi-agents Tracking. Here we presented tracking on single agent (although the agent can be in dynamic multi agent environment). Tracking in multi-agents environment is very challenging. First, we can utilize the information from group of agents. Second, tracking multi-agents raises naturally more hypotheses, and thus the observer needs to be selective.

Lossy observations. In Chapter , we suggested solution to the problem of lossy observations. As mentioned before lossy observations are the case that some features can not be seen in some point of time, for example due to hardware failures. In the literature there is an hidden assumption that all features can be observed. There are few works that tried to deal with the problem (e.g,[9]), but even in these works, just limited cases can be dealt with, cases that actions were observed without having seen previous actions that must be preliminary, or state of the world has changed without seeing actions that can explain these changes. Moreover, there is not any investigations on how efficiently these methods work, and what is the accuracy when there are missing observations. In future work we intend to implement the solution we suggested on section , and to explore more deeply the lossy observations case. We also intend to test it both on synthetic data and on dynamic, complex domains, such as ModSaf and RoboCup.

Probabilistic recognition. A limitation in our symbolic algorithms is that sometimes, more than one hypothesis is recognized, and we can not tell which hypothesis is more probable. There are some cases that some behaviors are more probable than others, for example when a person is going to the bank, it is more probable that he is going to do actions in his account, and not to rob the bank. There are behavior recognition systems that have the ability to rank hypotheses. However, these methods can not take into account state of the world, ordering between behaviors, lossy observations and are not capable of working with large number of behaviors, for more details see section 2. In future work, we intend to investigate the possibility of incorporating probabilistic inferring and extend our algorithms to deal with ranking

hypotheses, while keeping all the advantages in our model.

Interleaved behaviors. Another limitation in our model is that it is not capable with copping with agent that pursuing multiple goals. In our model we considered just the cases that the agent finishes series of behaviors in order to finish one goal, and just then move on to pursuing another goal. In some domains this is the case, but in others agent can start with one goal, then move to another goal, and finally return to accomplish the first goal. To our knowledge of the literature, just few works can deal with interleaved behaviors [11].

Prediction. Currently we can answer to two key queries: (i) what does the agent do now? and (ii) What did the agent do until now?. A query we have not yet considered is the agent likely to do next, considering the states or actions it executed up to now. While there exists much work on prediction (e.g., in the context of unix command-line predictions), this work is not integrated with plan-recognition work. Such integration would be interesting.

Suspicious behavior. We intend to apply our model to recognizing suspicious behaviors, i.e., behaviors that are not recognized exactly by the behavior library.

Bibliography

[1] J. F. Allen and C. R. Perrault. Analyzing intentions in utterances. *Artificial Intelligence*, 15:81–115, 1980.

[2] P. Bakker and Y. Kuniyoshi. Robot see, robot do: an overview of robot imitation. In *the AISB Workshop on Learning in Robots and Animals*, Brighton, UK, 1996.

[3] T. Balch. *Behavioral Diversity in Learning Robot Teams*. PhD thesis, Georgia Institute of Technology, 1998.

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

[5] E. Charniak and R. P. Goldman. A probabilistic model of plan recognition. In *AAAI-91*, 1991.

[6] E. Charniak and R. P. Goldman. A Bayesian model of plan recognition. *AIJ*, 64(1):53–79, Nov. 1993.

[7] P. R. Cohen, C. R. Perrault, and J. F. Allen. Beyond question answering. In W. G. Lehnert and M. H. Ringle, editors, *Strategies for Natural Language Processing*, pages 245–274. Erlbaum, Hillsdale, NJ, 1982.

[8] R. J. Firby. An investigation into reactive planning in complex domains. In *AAAI-87*, 1987.

[9] C. W. Geib and R. P. Goldman. Plan recognition in intrusion detection systems. In *In DARPA Information Survivability Conference and Exposition (DISCEX)*, June 2001.

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

[11] R. P. Goldman, C. W. Geib, and C. A. Miller. A new model of plan recognition. In *UAI-1999*, Stockholm, Sweden, July 1999.

[12] L. D. Goodman, B.A. On the interaction between plan recognition and intelligent interfaces. *User Modeling and User-Adapted Interaction*, 2:83–115, 1992.

[13] K. Han and M. Veloso. Automated robot behavior recognition applied to robotic soccer. In *Proceedings of the IJCAI-99 Workshop on Team Behavior and Plan-Recognition*, 1999. Also appears in Proceedings of the 9th International Symposium of Robotics Research (ISSR-99).

[14] J. Hong. Goal recognition through goal graph analysis. *JAIR*, 15:1–30, 2001.

[15] M. J. Huber and E. H. Durfee. Deciding when to commit to action during observation-based coordination. In *ICMAS-95*, pages 163–170, 1995.

[16] M. J. Huber, E. H. Durfee, and M. P. Wellman. The automated mapping of plans for plan recognition. In *Proceedings of UAI-94*, 1994.

[17] G. A. Kaminka and M. Bowling. Robust teams with many agents. In *AAMAS-02*, 2002.

[18] G. A. Kaminka, D. V. Pynadath, and M. Tambe. Monitoring teams by overhearing: A multi-agent plan recognition approach. *Journal of Artificial Intelligence Research*, 17, 2002.

[19] G. A. Kaminka and M. Tambe. Robust agent teams via socially-attentive monitoring. *Journal of Artificial Intelligence Research*, 12, 2000.

[20] H. A. Kautz. In P. Cohen, J. Morgan, and M. Pollack, editors, *Intentions in Communications*, chapter A Circumscriptive Theory of Plan Recognition. MIT Press, 1990.

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

[22] Y. Kuniyoshi, S. Rougeaux, M. Ishii, N. Kita, S. Sakane, and M. Kakikura. Cooperation by observation—the framework and the basic task patterns. In *the IEEE International Conference on Robotics and Automation*, pages 767–773, San-Diego, CA, May 1994. IEEE Computer Society Press.

[23] E. Kutluhan, J. Hendler, and D. Nau. A sound and complete procedure for hierarchical task network planning. 1994.

[24] S. Lenser, J. Bruce, and M. Veloso. CMPack: A complete software system for autonomous legged soccer robots. In *Agents-01*, pages 204–211. ACM Press, May 2001.

[25] J. F. Litman D., Allen. A plan recognition model for subdialogues in conversation. *Cognitive Science*, 11:163–200, 1987.

[26] M. J. Mataric. *Interaction and Intelligent Behavior*. PhD thesis, Massachusetts Institute of Technology, 1994.

[27] T. M. Mitchell. *Machine Learning*. McGraw-Hill, 1997.

[28] A. Newell. *Unified Theories of Cognition*. Harvard University Press, Cambridge, Massachusetts, 1990.

[29] M. Nicolescu and M. J. Mataric. A hierarchical architecture for behavior-based robots. In *AAMAS-02*, pages 227–233, Bologna, Italy, July 15–19 2002.

[30] D. V. Pynadath and M. P. Wellman. Probabilistic state-dependent grammars for plan recognition. In *UAI-2000*, pages 507–514, 2000.

[31] G. Retz-Schmidt. Recognizing intentions, interactions, and causes of plan failures. *User Modeling and User-Adapted Interaction*, 2:173–202, 1991.

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

[33] C. Schmidt, N. Sridhan, and J. Goodson. The plan recognition problem: an intersection of psychology and artificial intelligence. *Artificial Intelligent*, 11:45–83, 1978.

[34] M. Tambe, J. Adibi, Y. Al-Onaizan, A. Erdem, G. A. Kaminka, S. C. Marsella, and I. Muslea. Building agent teams using an explicit teamwork model and learning. *AIJ*, 111(1):215–239, 1999.

[35] M. Tambe, W. L. Johnson, R. Jones, F. Koss, J. E. Laird, P. S. Rosenbloom, and K. Schwamb. Intelligent agents for interactive simulation environments. *AI Magazine*, 16(1), Spring 1995.

[36] M. Tambe, G. A. Kaminka, S. C. Marsella, I. Muslea, and T. Raines. Two fielded teams and two experts: A robocup challenge response from the trenches. In *IJCAI-99*, volume 1, pages 276–281, August 1999.

[37] M. Tambe, D. V. Pynadath, N. Chauvat, A. Das, and G. A. Kaminka. Adaptive agent integration architectures for heterogeneous team members. In *ICMAS-00*, pages 301–308, Boston, MA, 2000.

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

[39] G. Weiss, editor. *Plan Recognition in Natural Language Dialogue*. the MIT Press, 1990.