Evaluating the SBR Algorithm using Automatically Generated Plan Libraries

Giovani Farias, Lucas Hilgert, Felipe Meneguzzi and Rafael H. Bordini
Pontifical Catholic University of Rio Grande do Sul – PUCRS – Porto Alegre, Brazil
Email: giovani.farias@acad.pucrs.br, lucaswhilgert@gmail.com, {felipe.meneguzzi, rafael.bordini}@pucrs.br

***Abstract*—Most approaches to plan recognition are based on manually constructed rules, where the knowledge base is represented as a plan library for recognising plans. For non-trivial domains, such plan libraries have complex structures representing possible agent behaviour to achieve a plan. Existing plan recognition approaches are seldom tested at their limits, and, though they use conceptually similar plan library representations, they rarely use the exact same domain in order to directly compare their performance, leading to the need for a principled approach to evaluating them. Thus, we develop a mechanism to automatically generate arbitrarily complex plan libraries which can be directed through a number of parameters, in order to create plan libraries representing different domains and so allowing systematic experimentation and comparison among the several plan recognition algorithms. We validate our mechanism by carrying out an experiment to evaluate the performance of a known plan recognition algorithm.**

I. Introduction

Plan recognition systems generally require a knowledge base that encodes into recipes the ways in which agent goals can be achieved. Such knowledge bases are represented as plan libraries with an often complex structure. The earliest plan libraries encoded recipes as collections of preconditions, subgoals, constraints, and effects [3]. Since then, many different algorithms have been used to perform plan recognition based, for example, on Bayesian networks [2], graph covering [6], and probabilistic state dependent grammars [9]. These methods typically use an individual plan library to represent the set of plans that are expected to be recognised, and the sequence of observations are matched against this library to generate recognition hypotheses ranked according to some rating method. While an agent performs some task, the observation sequence acquired is matched against the plan library, and the obtained sequences are processed as plan recognition hypotheses.

Plan recognition algorithms may be used in real world applications, which can have complex multi-feature observations, presenting a high computational cost to matching these observations against all possible plan steps in the plan library. However, there is no approach to automatically generate complex domain structures with a particular parameter set to simulate real-world complex scenarios, supporting experiments to evaluate the performance of various plan recognition algorithms. Furthermore, plan libraries often have a complex structure, due to the large number of possible observation sequences that need to be encoded, which makes it extremely laborious to create these libraries with the desired characteristics by hand. Thus, it is important to have a mechanism for automatically generating these structures in order to evaluate the plan recognition algorithms under varying conditions.

We develop an approach for automatic generation of plan libraries, as well as the observations sequences which serve as input for the plan recognition algorithms. These generated plan libraries and observations sequences can be used as test suites in experiments for practical performance evaluation. The contributions in this paper are as follows. First, we develop a plan library generator that allows the construction of such test suites based on a set of parameters that will determine the plan library complexity. Second, we create an input set generator to obtain observations sets from these libraries allowing us to use the same domain information to directly compare performance among various plan recognition approaches using different structures of plan libraries. Third, we validate the approach by generating and carrying out experiments to evaluate the performance of the *Symbolic Plan Recognition* (SBR) method proposed in [1].

This paper is organised as follows. Section II briefly surveys the area of plan recognition and the definition of *Symbolic Plan Recognition* (SBR) method. We describe the parameters used by the algorithm to create random plan libraries in Section III. We describe our algorithm to generate plan libraries based on given parameters in Section IV, as well as the algorithm to generate sequences of observations in Section V. Finally, we describe the experiments that validate the usability of our approach to in Section VI, and conclude the paper in Section VII.

II. Plan Recognition

Plan recognition refers to the problem of inferring one or more subjects' goals based on a set of observed activities by constructing a plan (or multiple plans) that contains them [6]. Algorithms to recognise the intentions and plans executed by autonomous agents have been studied for a long time in Artificial Intelligence under the general term of *plan recognition*. Such work has yielded a number of approaches to plan recognition [10], [8] and models that use them in specific applications [12], [7]. Kautz and Allen [6] focus on symbolic methods providing a formal theory of plan recognition, usually these approaches specify a plan library as an action hierarchy in which plans are represented as a plan graph with top-level actions as root nodes, and plan recognition is then reduced to a graph covering problem, so the plan recognition process attempts to find a minimal set of top plans that explain the observations. Symbolic approaches are a plan recognition mechanism that narrows the set of candidate intentions by eliminating the plans that are incompatible with current agent actions. Generally, these approaches assume that the observer has complete knowledge of the agent's possible plans and goals, handling the problem of plan recognition by determining which set of goals is consistent with the observed actions. The inputs to a plan recogniser are generally a set of goals which the recogniser expects the agent to carry out in the domain, a set of plans describing the way in which the agent can reach each goal, and a sequence of actions observed by the recogniser. The plan recognition [1] process itself consists in inferring the agent's goal, and determining how the observed actions contributes to reach it.

A. Symbolic Plan Recognition – SBR

The *Symbolic Plan Recognition* (SBR) approach [1] is a method for complete symbolic plan recognition that uses a plan library, which encodes agent knowledge in the form of plans. The SBR extracts coherent hypotheses from a multi-featured observation sequence using a *Feature Decision Tree* (FDT) to efficiently match these observations to plan steps in a plan library. A plan library is a knowledge base that codifies in some way the agent's beliefs concerning how the agent can reach each particular goal in the domain. Typically, a plan library has a single dummy root node, where its children are *top-level* plans and all other nodes are simply *plan steps*. In the library, sequential edges specify the expected temporal order of a plan execution sequence and decomposition edges link sub-steps which are an elaboration (expansion) of the given plan step. Each plan step has an associated set of observation features status of the agent and its actions, when these features status are met, the observations match a particular plan step.

The first stage of the SBR approach is the matching phase, in which the observations made by the recogniser are matched against plans in the plan library. The SBR algorithm considers complex observations presuming that each plan step has a set of feature status on observable features associated with it. When these feature status hold in regards to observed features of action execution (and in the correct order in case of sequential edges), the current observation is said to match that plan. Matching observations to plans can be computationally expensive if all plans are checked, and for each plan, all observed features are also checked (e.g., [5]). To speed-up this process of matching observations to plans, SBR augments the plan library with a FDT data structure, which efficiently maps observations to matching nodes in the plan library. An FDT is a decision tree, where each node represents an observable feature and each branch represents one possible value of this feature. Determining all matching plans from a set of observations features is efficiently achieved by traversing the FDT top-down until a leaf node is reached, because each leaf node is a pointer to a plan step in the plan library.

III. Plan Library Generator – Parameters

The algorithms were created in order to enable systematic analysis and performance comparison between several plan recognition algorithms given the variety of possible plan libraries. So, the *Plan Library Generator* (Section IV) produces plan libraries based on various given parameters, and the *Input Set Generator* (Section V) creates sequences of observations given a plan library. The parameters used by the *Plan Library Generator* to create random plan libraries are as follows:

Number of top-level plans (np): represents the branching factor of the root node, in other words, the number of children for root node. This value refers to the number of different independent top-level plans in a plan library.

Depth (dt): corresponds to the depth of plan trees. This depth value for the plan library (from the root node) determines the number of plan steps that a plan instance contains.

Minimum number of branches (mi): defines the minimum number of branches that all nodes (other than root) must have. This value must be in the interval $[1; ma]$.

Maximum number of branches (ma): represents the maximum number of branches that all nodes (other than root) may have, and this value must be greater than or equal to the *minimum number of branches* (mi).

Number of features (fs): defines the number of observable features available to be associated with a given plan step. Features are properties associated with the action represented by a given plan step, that need to be observed by the plan recognition algorithms for them to be able to recognise the execution of this particular plan step.

Number of features per node (fn): defines the number of features associated with each node. As a restriction, *number of features* (fs) must be greater than (or equal to) the *depth* (dt) multiplied by the *number of features per node* (fn). Thus, there are at least fn distinct features for each plan step belonging to a single plan instance.

Feature status (st): represents the number of values that may be associated with the features. Features with a specific status allow the identification of determined plan steps (actions) being executed by the observed agent. This value must be greater than or equal to zero ($st \geq 0$).

Sequential edges (sq): value in the interval [0, 1], which determines the probability of a branch to be created as sequential type. Thus, for example, $sq = 0$ means that all branches will be of decomposition type, whereas $sq = 1$ means that all branches will be sequential.

Duplication (pd): percentage of top-level plans that are duplicated in order to generate ambiguous paths. Each plan has an unique identification. However, plans with the same set of associated features and value of these features are similar because they will match the same set of observations, at least up to a point hence leading the recogniser to have multiple unresolved hypotheses.

IV. Plan Library Generator – Algorithm

The generation of the plan library is conducted as briefly described in Algorithm 1 (presented in Farias et al. [4]).

Algorithm 1 GenerateTree

```
Input: $np$, $dt$, $mi$, $ma$, $sq$, $fs$, $fn$, $st$, $pd$
Output: root node                                ▷ tree with root node
1: create root node
2: create top-level plans
3: for each top level plan do
4:     CreateBranches                  ▷ briefly described in Algorithm 2
5:     add children nodes
6: end for
7: duplicate plans             ▷ duplicate top-level plans according to pd
8: return root node
```

[1]For a good overview of plan recognition in general, see Carberry [3], and for the most recent research in the field of plan, intent, and activity recognition, see Sukthankar et al. [11].

Algorithm 2 `CreateBranches`

```
Input: $id$, $fs$, $pf$, $cd$
Output: node                                              ▷ branch starting from node
 1: create a node
 2: if $pf$ is not empty then
 3:     node get parent features
 4: end if
 5: if current depth $cd$ is equal to depth $dt$ then      ▷ node is a leaf
 6:     return node
 7: end if
 8: add sequential children
 9: add decomposition children
10: return node
```

Algorithm 1 starts by creating the plan library root node (Line 1), a decomposition node that is responsible for connecting all agent top-level plans and has no features assigned to it. The next step consists of creating top-level nodes (Line 2), which correspond to agent plans (e.g., plans "*p1*" and "*p2*" in Figure 1). These nodes are created as simple decomposition nodes to which no features are assigned. After the creation of top-level plans, the next step (Lines 3–6) consists in the creation of their respective branches as described in Algorithm 2 [4]. Finally, after the creation of individual top-level plans, the algorithm selects (Line 7) the ones that will be duplicated if a duplication percentage is non-zero.

Algorithm 2 describes the creation of plan-step nodes and their respective branches, receiving as input: the *new node identification* (id), *number of features* (fs), subset of features assigned to its *parent node* (pf), and the *current depth* (cd) in which the node is going to be created. Some parameters, such as: *number of features per node* (fn), *number minimum of branches* (mi), *number maximum of branches* (ma), *depth* (dt), and *sequential edges* (sq) are assumed as global. Algorithm 2 starts by creating a new node (Line 1) and checking whether the set of parent features (Line 2) is not empty, in which case the new node is a decomposition node and inherits the parent features (Line 3). In next step, before creating new node branches, algorithm checks if current level has reached the expected depth (Line 5). So, if the expected depth has been reached, new node is returned and plan path creation is completed, otherwise, if the depth has not been reached yet, algorithm goes to next step creating the next level of tree. The next stage in algorithm execution is to create the sequential branches of node. This step creates recursively all sequential nodes and each node created is added to the sequential children set of the new node (Line 8). After creating sequential branches, this algorithm uses a similar step to create decomposition branches of the new node (Lines 9).

The main difference between creating sequential and decomposition branches is the feature distribution among nodes. While creating sequential nodes the construction of feature subset is based on the whole feature set (fs), creating decomposition nodes is based on a subset of the feature set, which eliminates features already used by parent nodes. In Figure 1, nodes represent plans (first level) and plan steps (second level and below) of the plan library, and edges represent relations between them, where sequential links are represented by dashed arrows and decomposition links are represented by solid arrows. The *"root"* node is not considered a plan, being used only as a way of connecting various plans.

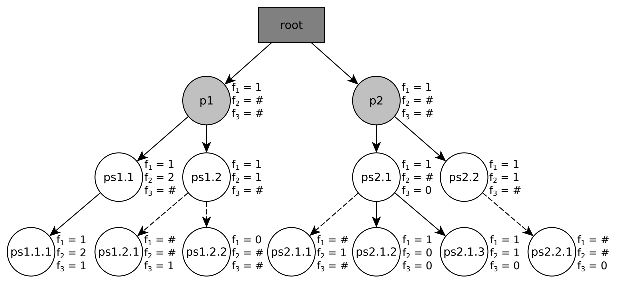
root
p1
$f_1 = 1$ $f_2 = \#$ $f_3 = \#$
p2
$f_1 = 1$ $f_2 = \#$ $f_3 = \#$
ps1.1
$f_1 = 1$ $f_2 = 2$ $f_3 = \#$
ps1.2
$f_1 = 1$ $f_2 = 1$ $f_3 = \#$
ps2.1
$f_1 = 1$ $f_2 = \#$ $f_3 = 0$
ps2.2
$f_1 = 1$ $f_2 = 1$ $f_3 = \#$
ps1.1.1
$f_1 = 1$ $f_2 = 2$ $f_3 = 1$
ps1.2.1
$f_1 = \#$ $f_2 = \#$ $f_3 = 1$
ps1.2.2
$f_1 = 0$ $f_2 = \#$ $f_3 = \#$
ps2.1.1
$f_1 = \#$ $f_2 = 1$ $f_3 = \#$
ps2.1.2
$f_1 = 1$ $f_2 = 0$ $f_3 = 0$
ps2.1.3
$f_1 = 1$ $f_2 = 1$ $f_3 = 0$
ps2.2.1
$f_1 = \#$ $f_2 = \#$ $f_3 = 0$

Fig. 1. Ex. plan library tree created by the *Plan Library Generator* ($np = 2$, $dt = 3$, $mi = 1$, $ma = 3$, $sq = 0.5$, $fs = 3$, $fn = 1$, $st = 2$, and $pd = 0$).

V. Input Set Generation

For the experiments in this paper, the set of observations used as input for the plan recogniser were automatically built using the generated plan library. In the input set, observations are organised in subsets known as *"queries"*, each query contains the necessary observations (one or various) for the recognition of a given plan. The number of observations in a query varies based on the structure of the plan to be recognised, for example, in Figure 1 the top plan *p1* can be recognised through *ps1.1.1* with a single observation, or through *ps1.2.2* using at least two observations given the temporal restriction between *ps1.2* and *ps1.2.2*. The creation of the observation sets is conducted automatically, as described in Algorithm 3 that receives as input a plan library and the number of queries to be generated.

Algorithm 3 `QueryGeneration`

```
Input: plan library $pl$, number of queries $nq$
Output: list $qs$ of sequences of observations
 1: $qs \leftarrow \emptyset$
 2: for $i = 0$ to $nq$ do
 3:     $pn \leftarrow$ getRandomPlan($pl$)
 4:     $obs \leftarrow$ getObservations($pn$)
 5:     qs.add($obs$)
 6: end for
 7: return list $qs$
```

The `QueryGeneration` (Algorithm 3) starts randomly selecting a plan of plan library to be recognised (Line 3). The chosen plan structure is then traversed and the necessary features (and feature values) for generation of observations are collected from its plan steps (process described in Algorithm 4). The generation process (Lines 2–6) is repeated until the number of expected queries is reached.

The `GetObservations` (Algorithm 4) describes how branches of a given plan are traversed. Initially (Line 2), each branch of a given node (both *sequential* and *decomposition*) is added to a single list. After that, one of these branches is randomly selected (as shown in Line 8). If the selected branch is of *decomposition* type the method is recursively called using its corresponding node (Line 11). For paths of plans composed only of decomposition branches (*e.g.*, "*p1*" to "*ps1.1.1*"), the method keeps being recursively called until a *leaf-node* (*e.g.*,"*ps1.1.1*") is found. Then, all features and feature values contained in the *leaf node* are used to generate an observation. This strategy is possible because in a *decomposition* relation the child node always inherits the features of its parent node. If

Algorithm 4 `GetObservations`

```
Input: node $nd$
Output: list $obs$ with a sequence of observations
 1: $obs \leftarrow \emptyset$
 2: $br \leftarrow$ getBranches($nd$)
 3: if $br$ is empty then
 4:     $ob \leftarrow$ buildObservation($nd$)
 5:     $obs.add(ob)$
 6:     return list $obs$
 7: end if
 8: $rn \leftarrow$ getRandomNode($br$)
 9: $db \leftarrow$ getDecBranches($nd$)
10: if $rn$ in $db$ then
11:     $ob \leftarrow$ getObservations($rn$)
12:     $obs.add(ob)$
13: else
14:     $po \leftarrow$ getPreviousObservations($nd$)
15:     $obs.add(po)$
16:     $ob \leftarrow$ getObservations($rn$)
17:     $obs.add(ob)$
18: end if
19: return list $obs$
```

the selected branch is a *sequential* branch, before the algorithm continues to follow the selected path, it has to generate an observation using the *features* of the current *plan step*. This procedure is necessary because a *sequential* branch represents a temporal relation in which for a plan step B to be validated, a plan step A has to be recognised first. For example, given the plan library shown in Figure 1 and the assumption that plan steps "*ps2.2*" and "*ps2.2.1*" correspond, respectively, to the actions "*get the ball*" and "*kick the ball*", for one to be able to execute the action "*kick the ball*", one has to execute the action "*get the ball*" first.

The procedure of building the previous observation is shown in Algorithm 5. In this algorithm, before the execution continues to follow through a sequential path it tries to build the observation using the *features* of the current node. In this situation, two cases have to be considered. In the first case, the current node is a *leaf node* (*i.e.*, has no decomposition children), thus the observation is built using the features and values of the current node (Lines 2–6). In the second case, the current node is a decomposition node, so a strategy is used in which the algorithm follows through the decomposition branches of the children nodes until a *leaf node* is reached (Lines 7–9). It is worth noting that the next decomposition node to be consulted is randomly chosen, as shown in (Line 8). After the previous observation has been created, the execution of Algorithm 4 is resumed and it continues to follow the selected path through the chosen *sequential* node (Line 16).

Algorithm 5 `GetPreviousObservations`

```
Input: plan Node $nd$
Output: list $obs$
 1: $obs \leftarrow \emptyset$
 2: if isLeafNode($nd$) then
 3:     $ob \leftarrow$ buildObservation($nd$)
 4:     $obs.add(ob)$
 5:     return $obs$
 6: end if
 7: $db \leftarrow$ getDecBranches($nd$)
 8: $pn \leftarrow$ getRandomBranch($db$)
 9: $ob \leftarrow$ getPreviousObservations($pn$)
10: $obs.add(ob)$
11: return $obs$
```

VI. Experiments

The experiments aim to demonstrate the usability of the approach presented in this paper to generate parametrised test structures, which allows principled performance evaluation for plan recognition approaches. An extensive set of experiments varying a number of parameters was carried out in order to evaluate the performance of the *Symbolic Plan Recognition* (SBR) algorithm, described in [1]. The *Plan Library Generator* (Section IV) builds a plan library based on the given parameters and the *Input Generator* (Section V) generates sequences of observations based on this plan library. The algorithms were implemented in Java SDK 1.7 (build 1.7.0_65-b17) and we ran the experiments on a Mac Pro Server (OS X 10.9.4) with two 6-core Intel Xeon (2.4 GHz) CPU, 32 GB of RAM (DDR3 1333MHz), and 2 TB of disk storage. In all experiments we evaluated the performance of SBR varying some parameters and for each of these values we generated a set of 200 random observation based on the given plan library. The average runtime of SBR matching those 200 observations are shown in Figures 2, 4, 6, and 8. The FDT training times for each experiment are shown in Figures 3, 5, 7, and 9.

The first experiment (Figures 2 and 3) was performed by varying the depth (dt) [3; 10] and the sequential edge probability (sq) [0; 1]. The other values were fixed as: $np = 10$; $mi = 1$; $ma = 3$; $fs = 10$; $fn = 1$; $st = 2$ and $pd = 0$. Figure 2 shows that SBR is more efficient in domains where plan paths do not have an expected temporal order of execution, i.e., plan paths with few sequential edges. Another important aspect is that the longer the sequence of actions necessary to realise a plan, the greater is the time to recognise it. This is because depth has a strong influence on the size and complexity of the plan library. This highlights the possibility of varying the temporal structure of the plan library by controlling the number of sequential edges.

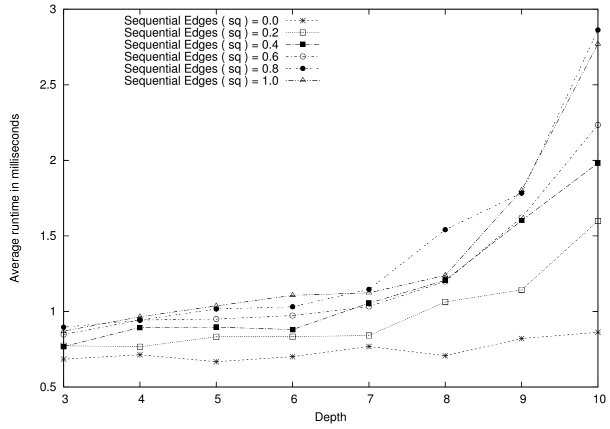

Fig. 2. Average matching runtime of SBR, varying dt and sq.

The time required to train the FDT is represented in Figure 3, where it is possible to observe that time increases according to the plan library depth (the deeper the plan library, the greater the FDT training time), and according to the number of sequential edges (the greater the number of sequential edges, the greater the training time).

Figure 4 shows the average runtime of SBR varying the number of top-level plans (np) [10; 100] and the number

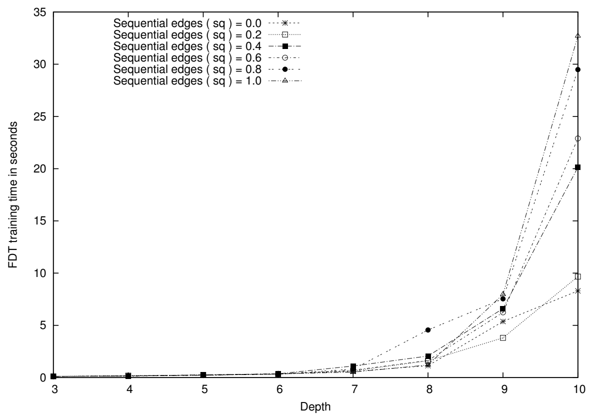

Fig. 3. FDT training time, varying dt and sq.

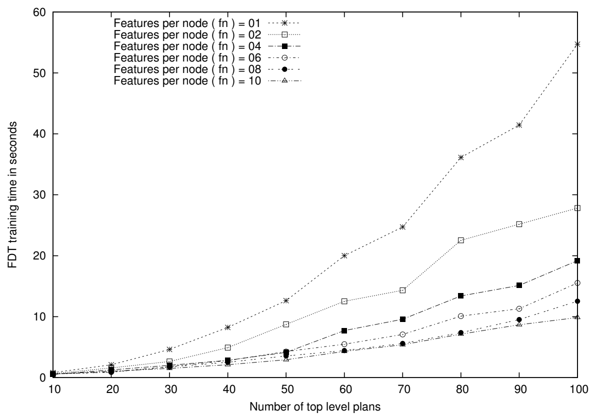

Fig. 5. FDT training time, varying np and fn.

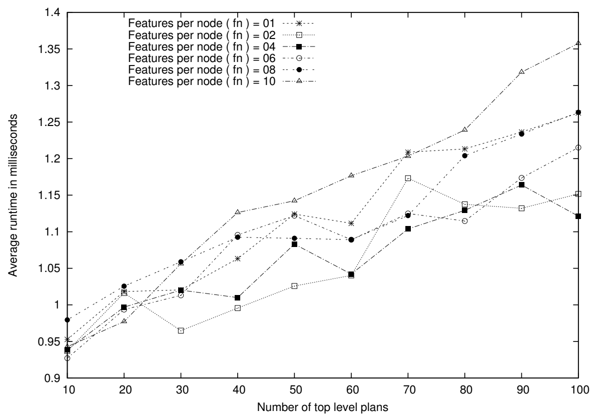

Fig. 4. Average matching runtime of SBR, varying np and fn.

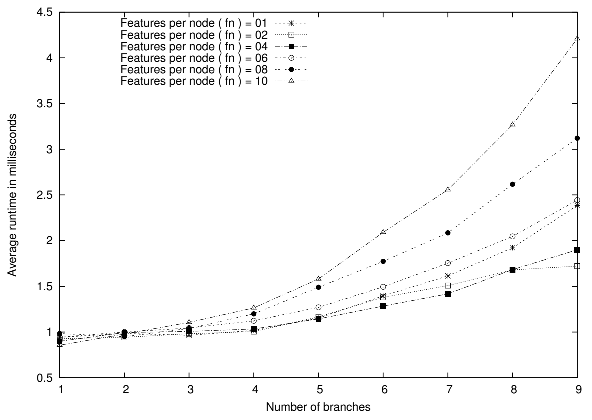

Fig. 6. Average matching runtime of SBR, varying ma and fn.

of features per node (fn) [1; 10]. The other values were fixed as: $dt = 5$; $mi = 2$; $ma = 2$; $fs = 50$; $st = 2$, $sq = 0.5$ and $pd = 0$. In this figure, we can observe that SBR recognition time tends to increase as the number of top-level plans increases in the domain. Figure 5 shows the influence that the number of top-level plans and the number of features per node carried on the FDT training time. We can observe that the increase in number of top-level plans increases the training time. This is contrary to what occurs with the number of features per node, where an increase in the number of features tends to lead to decrease in the time taken for training. This experiment indicates that it is possible to use these algorithms to generate increasingly complex plan-library structures to (stress) test various plan recognition algorithms.

The third experiment (Figures 6 and 7) was performed by varying the number of branches ($mi = ma$) [1; 9] and the number of features per node (fn) [1; 10]. It is worth noting that for this experiment, in particular, the number of branches represents the exact number of children of each node (i.e., parameters minimum and maximum number of branches assume the same value). The number of top-level plans (np) remains fixed at 10 and the duplication factor (pd) was fixed at 0.2, which presents the possibility of increasing the ambiguity in the plan library provided by the plan library generator algorithm. The other parameters assume the same values presented in the second experiment. Figure 6 shows that the time for SBR to recognise a plan increases as the number of branches and the number of features per node assumes higher values. This figure show a behaviour that is more regular than that seen in Figure 4, which may be due to the increased size and complexity of the plan library; such increase in size and complexity happens more quickly when we change the number of branches than when we change the number of top-level plans. The FDT training time presented in Figure 7 shows the major influence that branching factor (branches per node) has on SBR performance, especially for the highest values, although the time to recognise a plan remains at the scale of milliseconds (see Figure 6), the FDT training takes longer.

Finally, the fourth experiment (Figures 8 and 9) was performed with the aim of assessing the influence that the number of features per node (fn) and the feature status (st), i.e., the number of values that can be associated to the features, have on FDT training time. Figure 8 shows average runtime of SBR varying number of features per node (fn) [1; 10] and feature status (st) [1; 10]. The other values were fixed as: $np = 10$; $dt = 7$; $mi = 3$; $ma = 3$; $fs = 10$; $sq = 1$ and $pd = 0$. This experiment shows that the expected effect of using the FDT is diminished when the number of features per node (plan step) is set at 1, which essentially treats features as atomic. In Figure 9, we observe that the fewer features per node, the slower the FDT training time (the worst case is when there is only one feature per node). The range of feature status also influences the training time, where the greater the relative value of feature status, the smaller the training time.

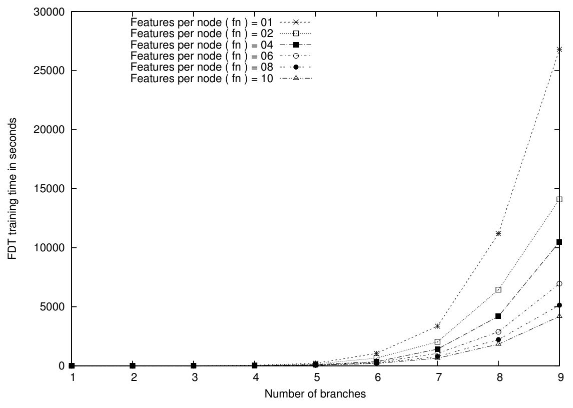

Fig. 7. FDT training time, varying ma and fn.

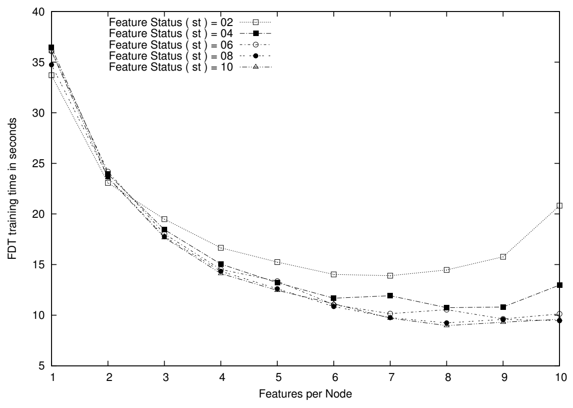

Fig. 9. FDT training time, varying fn and st.

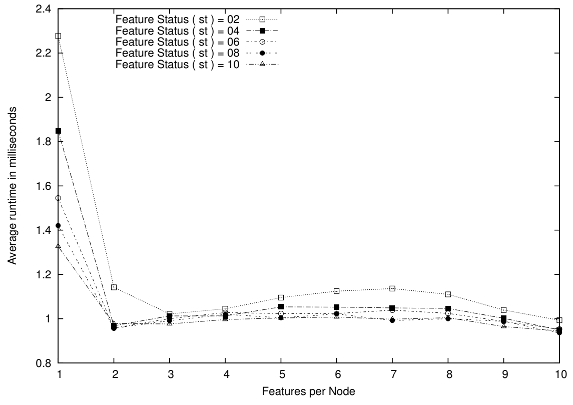

Fig. 8. Average matching runtime of SBR, varying fn and st.

VII. Conclusion

In this paper, we developed a framework that allows principled performance evaluation for plan recognition algorithms. A plan library generator was created to generate complex structures based on a number of parameters that will determine the complexity of the plan library. Thus, a unique representation of an information domain can be used to compare the performance of several plan recognition algorithms. This performance is directly related to the structure and size of the plan library, as well as the set of observations given to the plan recognition system. The size of a plan library is mainly determined by the number of top-level plans, the interval composed by the number minimum and maximum of branches, and by its depth. On the other hand, the ambiguity of plan library influences the amount of distinct plans that fit a given sequence of observations. In our approach, the amount of ambiguity is determined by: (i) the duplication parameter, which taking larger values implies more duplicated plans, thereby increasing the ambiguity; (ii) the number of features, as less features tend to decrease the possibility of a distinction between plans; (iii) the feature status, which assuming higher values enables greater distinction among plans that use the same set of features; and (iv) by the number of features per node where greater values cause more variety in plans.

Experiments were carried out to assess the effectiveness of this approach by generating several plan libraries with different structures to test the performance of the SBR algorithm. We observed that SBR has a better recognition time performance in domains presenting plan paths with fewer sequential edges and short sequences of actions and it presents worse performance when the number of branches and number of features per node assume higher values. Besides, SBR recognition time and FDT training time tend to increase according with the number of top-level plans in the domain. The FDT training time increases according to plan library depth and number of sequential edges, and decreases when the number of features per node grows. Finally, experiments show the capacity of our approach to create complex and varied structures, as well as the possibility to change the temporal structure and ambiguity of the plan library to evaluate plan recognition algorithms.

Acknowledgement: This paper was part of the research project entitled "Semantic and Multi-Agent Technologies for Group Interaction", sponsored by Samsung Eletrônica da Amazônia Ltda. under the terms of Brazilian federal law No. 8.248/91. This research was also supported by CAPES and CNPq. Felipe Meneguzzi thanks CNPq for the support within process numbers 306864/2013-4 under the PQ fellowship and 482156/2013-9 under the Universal project programs.

References

[1] D. Avrahami-Zilberbrand and G. A. Kaminka, "Fast and complete symbolic plan recognition," in *Proc. IJCAI*, 2005, pp. 653–658.

[2] H. Bui, "A general model for online probabilistic plan recognition," in *Proc. IJCAI*, 2003, pp. 1309–1315.

[3] S. Carberry, "Techniques for plan recognition," *User Modeling and User-Adapted Interaction*, vol. 11, no. 1-2, pp. 31–48, Mar. 2001.

[4] G. P. Farias, L. W. Hilgert, F. R. Meneguzzi, R. Vieira, and R. H. Bordini, "Automatic generation of plan libraries for plan recognition performance evaluation," in WI-IAT 2015, Singapore, 2015 - Volume II, pp. 129–132.

[5] G. A. Kaminka and M. Tambe, "Robust agent teams via socially-attentive monitoring," *JAIR*, vol. 12, no. 1, pp. 105–147, mar 2000.

[6] H. A. Kautz and J. F. Allen, "Generalized plan recognition," in *AAAI*, 1986, pp. 32–37.

[7] J. Oh, F. Meneguzzi, K. P. Sycara, and T. J. Norman, "Prognostic normative reasoning," *EAAI*, vol. 26, no. 2, pp. 863–872, 2013.

[8] R.F. Pereira, and F. Meneguzzi, "Landmark-based Plan Recognition," *Procs. ECAI*, (to appear), 2016.

[9] D. V. Pynadath and M. P. Wellman, "Probabilistic state-dependent grammars for plan recognition," in *Proc. AUAI*, 2000, pp. 507–514.

[10] M. Ramírez and H. Geffner, "Plan recognition as planning," in *Proc. IJCAI*, 2009, pp. 1778–1783.

[11] G. Sukthankar, R. P. Goldman, C. Geib, D. V. Pynadath, and H. H. Bui, Eds., *Plan, Activity, and Intent Recognition: Theory and Practice*. Elsevier, 2014.

[12] G. Sukthankar and K. Sycara, "Activity recognition for dynamic multi-agent teams," *ACM TIST*, vol. 3, no. 1, p. 18, 2011.