University of Aberdeen - Universitá degli Studi di Padova

Department of Computing Science - Department of Computer Engineering
Master Thesis in Computer Engineering

Norm Recognition in Multi-agents Systems

Candidate:
Alberto Lazzarin

Supervisor at University of Aberdeen:
Prof. Nir Oren

Home Supervisor:
Prof. Carlo Ferrari

Academic Year 2012-2013

Abstract

The purpose of this work is to explore the recognition of norms in multi-agent societies through the use of plan-recognition. In particular the project proposes to verify the effectiveness of some techniques in recognizing prohibitions in an environment without full observability.

Following a description of our approach, we will describe a simulation, implemented in Java, in which some selected procedures presented in the first background section will be modified to fit in the presented environment (a simplified road network). Later tests on the prototype will be conducted to compare the plan recognition approach with another one, namely a violation identification approach.

Acknowledgements

I would like to thank my supervisor, Nir Oren, for his advice and help. Also, I would like to thank Dr Felipe Meneguzzi for his advice.

Contents

Introduction

Norm Learning is a recent area of research in Artificial Intelligence. A *norm* could be seen as a list of constraints or rules defined by an environment or, more generally, by a multi-agent system. Norms are not part of the agent's architecture and often have to be identified by an agent before he can act in an proper way for the system. Similarly to what has been stated in [19], the expected behaviour of agents in an environment is described by means of an (not always) explicit specification of norms. According to the definition given by the researchers involved in the NorMAS 2007, "*A normative multi-agent system is a multi-agent system organized by means of mechanisms to represent, communicate, distribute, detect, create, modify and enforce norms, and mechanisms to deliberate about norms and detect norm violation and fulfillment*" [13]. An interesting research area in this context is **Norm Learning**, namely the study of the learning mechanics that lead an agent to the discovery of new norms in an environment. Norms are essential to allow different kinds of agents to interact properly in the same environment and Norm Learning is essential to allow agents to adapt in a new environment already populated by other agents.

In a similar context (more generally in a multi-agent system) some agents could be interested in observing the actions of some other agents in the environment. The observing agents could be representative of some entities that do not know about a new environment or the observed agents themselves and want to learn more about all of this. **Plan recognition** is the task of recognizing the whole plan, or the whole sequence (past and future) of actions of one or more observed agents. Intention recognition is related to this topic, but consists of recognizing just the final goal of the observed entities. Plan recognition, instead of Norm Learning, has a long history and has developed in this last decade effective solutions to the problem. Considering that the real challenging effort is to learn norms (only) by observing agent's behaviour in a selected environment, an interesting and still unexplored area could be the use of plan-recognition techniques to infer norms. This project aims to explore this research area by implementing and evaluating an interesting Norm-Learning technique based on Plan Recognition.

The work we present will first review, in the Background Section, some of the most important works appearing in Plan Recognition and in Norm Learning. Then, in Section 2, we will describe a Java-based simulation of a road network. This environment is useful because it defines a potentially big plan library. Moreover if we map plan library's states into graph's nodes and state-transitions into graph's arcs, this environment can be adapted to a lot of situations. In our test-program we developed two norm learning procedures: one is based on a recent and effective plan recognition technique, the other is based on the detection of an agent's norm violation. In this work we evaluate the plan-recognition algorithm in inferring norms under many conditions and we compare it with violation-based norm identification. The test's results will be shown and commented in the Section 3.

Finally we will describe some possible future work in Section 4.

1 Background

This section provides the background for all work described in the report. This will be useful to understand the reasons behind the project. Some of the most interesting norm and plan recognition theories will be presented and then some of them will be the starting point for the next sections.

1.1 Plan Recognition

The observing agent, whose task is to recognize other agents' plan, could be interacting with the observed agents and, on the other hand, the observed agents could be aware of the observer. Cohen, et al. [1] and Geib [2] distinguished three classes of plan recognition problems.

keyhole case: The observed agents don't care about the observing agent or are not aware of it. The observed agents will never act against the observer or aim to hide their actions.

intended case: The observed agents are aware of the observing agent and act to help the observing agent in recognizing their action. A communication system (signals for example) is required. The observing agent could have an active role in the environment, in particular giving some help back to the observed ones to make them reach their goal more easily.

adversarial case: The observed agents are aware of the observing agent and hostile to them. They will/could act to hide their action or, more generally aim to make plan recognition more difficult.

Two assumptions are normally made when considering plan recognition.

The first assumption is that every observed agent acts to achieve a goal. However an active agent could have partial or erroneous knowledge of some actions' result, so he could execute one or more actions which do not lead to its real goal and make the plan-recognition more difficult.

Secondly there is a plan library that is shared between the observing agents and the observer. A plan library specifies all the actions necessary to achieve one or more goals and the conditions which have to be satisfied in order to start and complete every action. Commonly the observer agent has full knowledge of the plan library [3]. However the observed agents could have only partial knowledge of it.

The observer, depending on his capabilities, can have full observability of the environment and (consequently) of the sequence of actions of the observed agent, but usually a third assumption is adopted, namely the unambiguous recognition of the observed agent's individual actions. This last assumption is not realistic but is useful in a simulated environment where most (or all) of the effort is put on the study of the plan recognition component.

Plan-Recognition techniques

One of the first and most popular techniques appearing in the plan/intention recognition literature is *abduction*. This is a reasoning process based on the "Affirming the consequent" fallacy which consists in asserting the validity of A starting from the rule "if A, then B" and from the validity of B. Obviously this could be seen as a form of "guess" and it could result in wrong conclusions. Moreover abduction usually provides more than one hypotheses explaining an observation in a context where the observer agent is supposed to choose just one, or a few, of them. Clearly the observer's chosen plan is among all the hypothesis provided by the abductive reasoning. The choice among multiple hypotheses could be done by a global or local criteria. A global criteria needs a form of universal ranking for every hypothesis[1]. Local criterias are based on different evaluation metrics which are associated with the rules of the background theory of the agent.

While a large number of plan recognizer algorithms have been proposed [4, 5, 7, 6, 8, 9, 10], (see Appendix C for further details) mostly based on a combination of abductive and probability reasoning, in this work we focus on the Symbolic Plan Recognizer, described next.

The Symbolic Plan Recognizer

In 2005 Dorit Avrahami-Zilberbrand and Gal A. Kaminka [11] introduced an efficient plan-recognition system based on a tree-representation of the plan library. This was shown to be faster than all of the algorithms previously presented. Furthermore this recognizer attempts to keep the current observations consistent with the history of the observed actions, avoiding a-priori inconsistent plans.

The plan-library is modeled as a tree-like structure (single-root directed acyclic connected graph). Every node corresponds to a state or a "plan-step" except for the root-node which has no particular meaning (in the paper it is called the *top-level plans* node). Edges could be of two types: vertical edges decompose plan-steps into sub-steps, sequential edges define a temporal order among all of the plan-step nodes. At any given time, the active agent is assumed to be executing a root-to-leaf path. Note that cyclic plans are allowed[2] but the graph must remain hierarchically acyclic. Figure 1 (taken from [11]) gives a simple example of such a structure. The top-level plans, in this particular case, are *defend*, *attack* and *score*. The node *score* is linked by a sequential edge to *attack* which means that an acting agent can choose the plan *score* only if he chose before the plan *attack*. A path which could be chosen by the agent is *root -> defend -> position -> turn -> with ball*. Note that "turn" has two vertical children in the graph: *with ball* and *without ball*. This means that when the *turn* plan-step is detected, the ball's position is (has to be) clearly known. In the described system an agent can interrupt its plan. For example, while it is executing *root -> defend -> position -> turn with ball -> clear -> position*,

[1] A simple example is to assign a probability to every hypothesis based on its frequency.
[2] plans made by repeated plan-steps

it can interrupt the plan and choose another one, like *root -> attack -> etc*, starting from the root node.

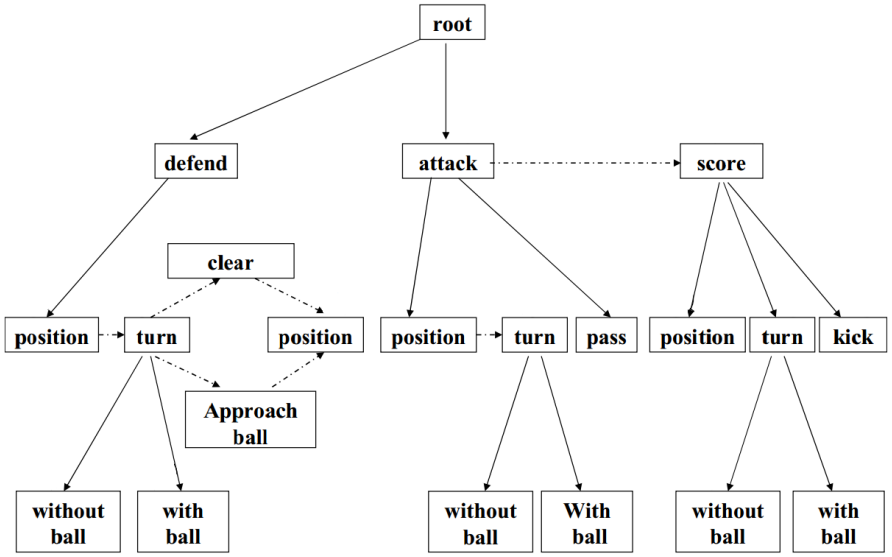


Figure 1: example of a SPR Plan Library

The first step of the plan recognition is always action recognition, namely the matching of a set of observations to the corresponding set of possible plan-steps.

More interesting is the second phase. After the initial matching, the observer-agent is provided with some possible plan-steps which refer to the actual set of observations. Every plan-step found could be part of more than one of top-level plans which is being executed by the actor. However not every plan is compatible with the past set of observations, that's why a consistency checking is required (at any given time, only one top-level plan can be executed). This is made by the CSQ (Current State Query) which is reported below.

Algorithm 1 CSQ(Matching results M, Library g, Time-stamp t)

1: **for all** $v \in M$ **do**
2: $PropagateUp(v, g, t)$
3: **for all** $v \in M$ **do**
4: **while** $tagged(v, t) \wedge \neg\exists ChildTagged(t)$ **do**
5: $delete\_tag(v, t)$
6: $v \leftarrow parent(v)$

Algorithm 2 PropagateUp(Node w, Plan Library g, Time-stamp t)

1: $Tagged \leftarrow \emptyset$
2: $propagateUpSuccess \leftarrow true$
3: $v \leftarrow w$
4: **while** $v \neq root(g) \wedge propagateUpSuccess \wedge \neg tagged(v,t)$ **do**
5: **if** $tagged(parent(v),t) \vee features(parent(v)) = \emptyset$ **then**
6: **if** $tagged(v, t-1) \vee \exists PreviousSeqEdgeTaggedWith(v, t-1) \vee NoSeqEdges(v)$ **then**
7: $tag(v,t)$
8: $Tagged \leftarrow Tagged \cup \{v\}$
9: $v \leftarrow parent(v)$
10: $propagateUpSuccess \leftarrow true$
11: **else**
12: $propagateUpSuccess \leftarrow false$
13: **else**
14: $propagateUpSuccess \leftarrow false$
15: **if** $\neg propagateUpSuccess$ **then**
16: **for all** $a \in Tagged$ **do**
17: $delete\_tag(a,t)$

CSQ (Algorithm 1) calls, for every node which appears in the observations set at time t (Matching results **M**), the "PropagateUp" procedure. PropagateUp (Algorithm 2) traverses the graph from the current node v to the root of the graph by its ancestors (line 4) and uses time-stamps to tag nodes in the library that are consistent with the current and previous observations. PropagateUp assumes that the calls to it have been made in order of increasing (vertical) depth. This allows an assumption (line 5) that matching parents are already tagged or do not have any associated observable features. Moreover, for every scanned node, it checks if the current node is *temporally consistent* with the rest of the graph (line 6). A node at time t is temporally consistent with the rest of the graph if one of these conditions holds:

- The current node was tagged also at t - *1* (self-cycle is allowed); or
- the current node follows a sequential edge from a node tagged at t - *1*; or
- the current node is a first child, that is there is no sequential edge leading to it (plans interruption is allowed)

If a particular node w, is temporally consistent, then the algorithm tags it with t (lines 7 and 8) and repeats the same inspection with its parent, up until the root (line 9). Otherwise all the tags put on the nodes traversed in the process (including the initial one, v) are removed (lines 15, 16 and 17): a node

is temporally consistent only if all of its parents are (this could be seen as the fourth condition).

Finally CSQ removes the tags from the *hierarchically inconsistent* nodes from M, which are the nodes without any (vertical) child tagged.

SPR seems to fit in a fully observable environment where keyhole plan-recognition is performed. Thanks to its simplicity[3], and to the fact that it is almost independent from other plan-recognition theories, Symbolic Plan Recognition can be used in lots of situation and can be integrated in other more complex techniques. For example later work extends the symbolic plan recognizer with the addition of an utility function for the observer [12].

1.2 Norm-Identification

Norm-Identification or Norm Learning could be done through *passive learning* which consists in listening to some advice provided by a normative entity, or through *active learning*. Active learning, as suggested by Hamada et al. [14], can be divided in three categories:

1. **Experiential learning**: the agent involved in the learning process is active and can violate norms. At every violation the environment (or other agents) "punish" the agent. The agent should infer the norms from the rewards or sanctions resulting from its actions.

2. **Observational learning**: the agent involved in the learning process is not necessarily active (in the sense that it "acts" in the environment) but has to deduce norms by observing the actions and reactions of the active agents in the system. Notice that this particular kind of Norm-identification is very close to Plan-Recognition

3. **Communication-based learning**: the agent involved in the learning process is active in the sense that it has to send and receive messages or signals to the other agents in the system in order to learn (or simply memorize) norms. Although it is considered an *active-learning* technique, communication-based learning itself is close to passive learning. After "asking the question" the agent has to "listen to the answer".

Hybrid methods are obviously possible and could give better results as verified by Bastin Tony Roy Savarimuthu's work [15]. There he compared three approaches to norm-identification in a simple example, using an experiential learning approach; a combination of experiential learning and observation learning[4]; and a combination of all three. Not unexpectedly, the second approach performed better than the first, and the third was sightly better than the second.

Related to this, [16] describes an observation and communication learning approach without full observability of the environment. The observer agent, through observational learning, creates a set of candidate norms which will be

[3] authors call it "lazy-commitment" plan recognition.
[4] making use of association rule mining

later verified through an exchange of simple messages with the other agents. The observer is also provided with a utility-function which could make the agent violate a recognized norm. This aspect and the communication component are not relevant to our work because we concentrate on observational learning approaches. Every time an agent violates a norm by performing an action or a sequence of actions which are forbidden in the system, another agent, if it is geographically close to the violating one, can sanction it. The observer agent, which sees this event, starts its reasoning invoking the Candidate Norm Inference (CNI) algorithm which is shown below.

Algorithm 3: Candidate Norm Inference algorithm (main algorithm)

```
1 foreach invocation of the norm inference component do
2 |   Create event episodes ;                    /* see Algorithm 4 */
3 |   Prune event episodes ;                      /* see Algorithm 5 */
4 |   Create Candidate Norm List (CNL) ;          /* see Algorithm 6 */
5 end
```

Algorithm 4: Pseudocode to create Event Episode List

```
  Input: Event Sequence (ES), Window Size (WS)
  Output: Event Episode List (EEL)
1 foreach invocation of the norm inference component do
2 |   foreach special event in ES do
3 |   |   Create an Event Episode (EE) with the last n events that precede
  |   |   the special event where n=WS;
4 |   |   Store EE in EEL;
5 |   end
6 end
```

Algorithm 5: Pseudocode to create Pruned Event Episode List

```
   Input: Event Episode List (EEL), Unique Event Set (UES), Norm
          Pruning Threshold (NPT)
   Output: Pruned Event Episode List (PEEL)
   /* construct Unique Event Set (UES)                              */
 1 foreach event E in ES do
 2 |   if E ∉ Unique Event Set (UES) and E ∉ Special Event Set (SES)
   |   then
 3 |   |   Add E to UES;
 4 |   end
 5 end
   /* construct Removable Event Set (RES)                           */
 6 foreach event E in Unique Event Set (UES) do
 7 |   Calculate OP(E);
 8 |   if OP(E) ≥ NPT then
 9 |   |   Add event to Removable Event Set (RES);
10 |   end
11 end
   /* construct Pruned Event Episode List (PEEL)                    */
12 foreach Event Episode (EE) in EEL do
13 |   foreach event E in an EE do
14 |   |   if event E in RES then
15 |   |   |   Remove event from EE;
16 |   |   end
17 |   end
18 end
```

Algorithm 3: Every time the *norm inference algorithm* is invoked, namely when the observer notices a violation in the environment, Algorithm 4, 5 and 6 are called to infer norms.

Algorithm 4: the input is an Event Sequence (**ES**), the sequence of events which precede the invocation of the norm inference component (Algorithm 1), and a parameter (Windows Size, **WS**) which corresponds to the number of (recent) events, previous to the Special Event, which will be taken into consideration. Special events are violation-events. One of them could be, for example, an agent which yells at or punishes another agent. The output is the Event Episode List (**EEL**), a list of all the sequences of events which could have activated the special event. The algorithm scans the history and looks for all of the special events in it. For every Special

Event (line 2), the observer stores the last n events, where n is equal to WS, which precede the special event in a list which represents a potential prohibited sequence of actions (line 3). This list is later put into the Event Episode List (line 4).

Algorithm 5: the input is a EEL, the Unique Event Set (**UES**), the set of all possible distinct events in the environment, and the Norm Pruning Threshold (**NPT**), a number. The procedure inserts every new (unique) event found from the EEL in the UES (lines 1, 2 and 3). Note that it is inserted only if it is not a "Special Event" (i.e. an event that will activate the norm inference component[5]). Then the Occurrence Probability (which could be the frequency), **OC**, of every Unique Event (line 6) is calculated (line 7). If the OC of an UE is greater or equal than NPT (line 8), it will be removed later from every Event Episode (EE) (line 9 and lines from 12 to 15). The meaning of this choice is intuitively to discard every *frequent* event from the list of the possible norm violations.

Algorithm 6: One of the main, and most interesting, assumption of the original paper is that a violation could be due not only to a single event, but also a particular sequence of events. Algorithm 5 is therefore insufficient to complete the norm inference. Algorithm 4 is a modified version of the well known WINEPI algorithm [17] used in Data Mining to find "frequent" temporal sequences/sub-sequences. The output is all of the events, or the sequence of events, with length at most equal to WS, which could be a violation of that particular norm-based system. This particular algorithm can be object of study for future works. However we will not show it in detail because our work deals only with single-event prohibitions. For further details the reader is referred to [16].

1.3 Norm Identification and Plan Recognition

A different approach to norm recognition involving plan recognition [18] is perhaps one of the first to do this, describing a simple, plan-recognition based, algorithm to find out norms or, more in detail, prohibited or obligated states/actions. The approach operates in a partially observable domain and there is the strong assumption that every observed agent never violates a norm. The main idea is this: the observing agent creates some possible plans, without knowing norms, then detects the actors' plans with a plan recognition algorithm. The differences between the detected plan and the alternative plans made by the observer could correspond to some prohibitions/obligations which the observer is not yet aware of. The pseudo-code of the algorithm is reported below in Algorithm 7. The "traces" correspond to the recognized actions by the observer.

[5] for example, the sanctioning action.

Algorithm 7 The norm detection algorithm.

```
1:  function DETECTNORMS(Traces)
2:      potProhib = ∅, potOblig = all possible states
3:      notProhib = ∅, notOblig = ∅
4:      for all t ∈ Traces do
5:          p ← DETECTPLAN(t)
6:          g ← goal(p)
7:          AltPlans ← PLANNER(g)
8:          posStates ← {states(altPlan)|altPlan ∈ AltPlans}
9:          notProhib ← notProhib ∪ states(p)
10:         potProhib ← potProhib ∪ posStates\notProhib
11:         potOblig ← (potOblig ∩ states(p))
12:     return potOblig, potProhib
```

Initially the norm detection algorithm creates several empty sets of states (line 2 and 3): *potProhib* (potentially prohibited states), *notProhib* (states which are allowed for sure) and *notOblig* (states which are not obliged). It also creates the *potOblig* (states which are potentially obliged) which is initialized as the set of all the possible states of the plan library. For all the observed traces (line 4), the observer uses the plan-recognition component to detect (only) one possible plan (line 5). Then the planning component is called (line 7): the observer with its knowledge plans one or more alternative paths towards the same goal (line 6) as the detected. The *AltPlans* set is initialized with all of these alternative plans. The set *posStates* is then created (line 8): all of the states of all of the possible alternative plans go into this list. Intuitively if no acting agent violates a norm then the states from their recognized plans have to be allowed: the states of the detected plan are added to *notProhib* (line 9). Now if a state is in the *posStates* but it is not in *notProhib*, maybe the reason could be that the mentioned state is forbidden: all of the states which belong to *posStates* but are not in *notProhib* are added to *potProhib* (line 10). Finally all of the states, which have not been visited by actors during the plan, are removed from the potentially obligated states set (*potOblig*) (line 11).

In [18] few results are presented. It is only stated that the procedure detects some false positives, but "as the number of traces increased, these false positives vanished". Plan Recognition, in this work, is implicitly supposed to be perfect. In our experiment we will extend this work and see how Algorithm 7 performs with a Plan-Recognizer which may commit mistakes.

2 Experiment report

This section will show and explain the software which has been later used to test the Norm Learning algorithms. After describing the multi-agent environment, and its java-simulation, two different norm-identification methods will be adapted and then (third section) evaluated.

The prototype attempts to recreate two of the norm-inferring works, presented in the previous section, with their ideal assumptions but in a more universal and realistic system, namely a system with a large plan-library, with partial observability. Because of these new initial conditions, we modify some features of the original approaches.

2.1 The environment

For the project we need a basic environment where we can easily represent an agent's state and an agent's state-transition. In this case a *plan* can be defined as a path made of state-transitions which links a (initial) state to another one (destination) and a *norm* as an obliged or forbidden state-transition. Therefore we choose a directed graph with a set of nodes and a set of (oriented) arcs. All of the arcs have the same cost, which is equal to 1 for simplicity. So we define the optimal path or plan from a node to another as the path with the lowest number of arcs. Norms in these experiments are limited to prohibitions which are represented by forbidden arcs (forbidden state transitions). The observed agents are free to roam on this map. Their plan is simple: they decide on a node destination (different from their initial position) and find a path to reach it. After having gone to the final node, agents stay inactive for a while ("rest") and then they restart the process again.

The observer agent is inactive and has to find forbidden arcs only by observing actors' behaviour. Note that the **plan library**, that determines which nodes can be reached from a particular node, is **shared** between the actors and the observer. Put another way, the observer (with the exception of the forbidden arcs) and actors are completely aware of the road network. The observer may not have full observability of the agents' moves, i.e. he could see and record only some positions of the observed agents during their trip.

We are working in a **keyhole** plan-recognition system, similar to most of the works presented in the previous section.

Moreover the action-recognition component is ignored[6]. This last assumption allows our work to be adapted to almost every situation. In fact every node corresponds to a state of a particular agent and every arc can represent an action (or a set of actions) that leads to that state-transition. A simple example is shown in Figure 2.

[6] in other terms, there is the assumption that actions are recognized unambiguously

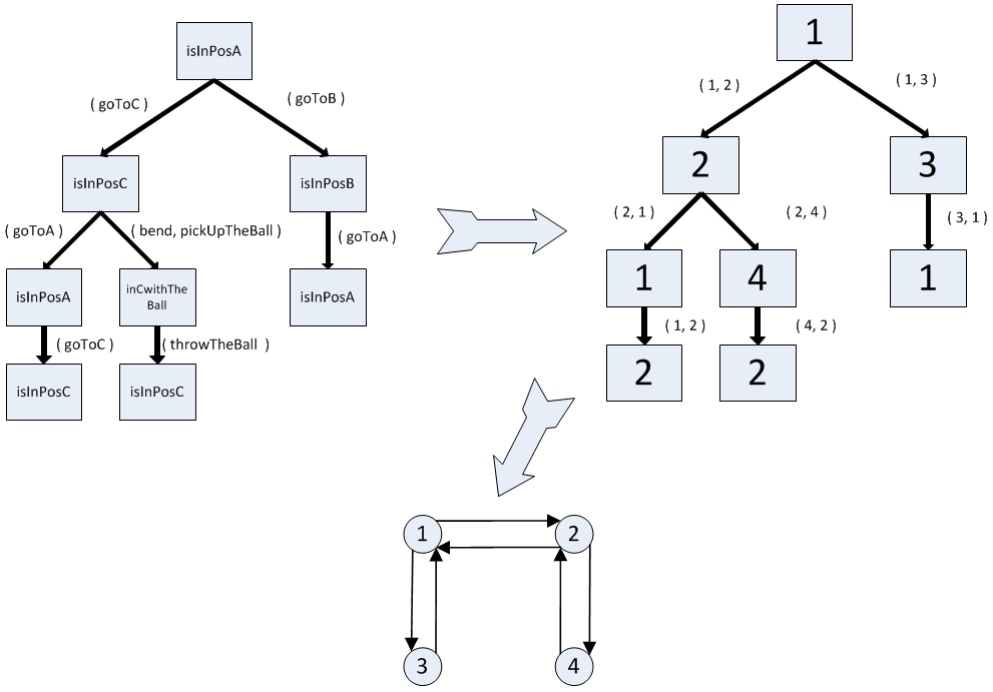


Figure 2: example of a (possible) conversion from a generic plan library to an oriented graph

For the project we created a graph, shown below in Figure 3, with 16 nodes and 30 arcs.

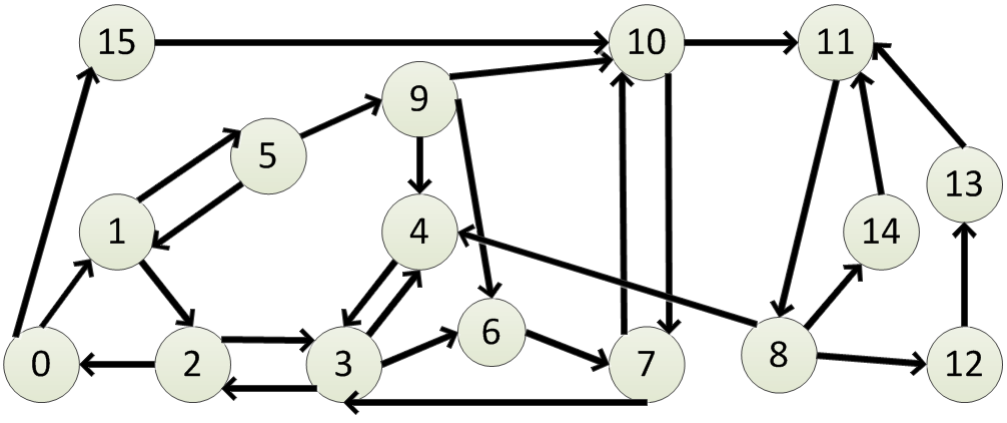


Figure 3: the environment (a road network)

All the 30 arcs are always allowed, but the prototype randomly generate 5

prohibited arcs every time it starts.

2.2 Implementation choices

We implement the project in Java[7]. In this work we will just evaluate the effectiveness of the algorithms in recognizing norms, and not their time-performance. Therefore language is not crucial and we choose Java due to the author's familiarity. We use also the Jason plugin to model the observed agents (actors). Jason is a java-based interpreter for an extended version of AgentSpeak(L)[8]. Its use in this work is due to its usefulness in modeling simple multiagent systems. Note that all norm-identification and plan-recognition algorithms are implemented exclusively in Java.

The graph-environment is a set of integers, which represents nodes, and a set of OrientedArcs. OrientedArc is a class made for this project which models an oriented arc of the graph. It is made of a tuple of integers and of a boolean (isAllowed) which indicates if this arc is prohibited or not.

The environment is populated by 4 agents: 3 actors and the observer agent. Every actor calls the *randPlan* method at the beginning. The procedure selects a random destination, different from the starting node, and then plans a path towards it. This is done by calling a modified version of the A* search: the path from the starting node to the destination may be not the optimal one, namely the path with less arcs. When a actor reaches its destination, it calls the *rest* method and waits.

The observer is an entity which cannot move and is made of 3 *ObservationTowers* and a *CognitiveCentre*. Each ObservationTower is a thread which just follows all the actors' movements, recording the list of nodes touched by the actor during its trip in an array. We have an ObservationTower for every observed agent.

Once all the agents have called the *rest* procedure, the CognitiveCentre is activated. This thread gets the (partial) data (the list with some of the agents' moves) taken from the three ObservationTowers as input and tries to detect prohibited arcs. At the end of the process, CognitiveCentre is paused, the 3 agents awake, and the system restarts and behaves in the same way again. Note that there is a simplification: the CognitiveCentre and the 3 actors should be temporally concurrent to make the simulation more realistic. However this compromise doesn't ruin the main task of the work in our view.

The final program runs as a Plan-Recognition prototype, where a Plan-Recognition approach is implemented and tested, and as a Violation based prototype, where a Violation-Identification approach is implemented and tested. Both approaches are tested on the same environment, including actors, but under some different assumptions. We will describe them in detail next.

[7] Java Development Kit v6 with Eclipse IDE
[8] for further information: http://jason.sourceforge.net/wp/

2.3 Plan-recognition based prototype

In our simulated environment we assume that actors **never** violate a norm, namely they never traverse a prohibited arc. This assumption make our system closer to the one described in [18]. Moreover it guarantees that if the plan-recognition component makes no mistake (the *base case*), there will never be false negatives (prohibited arc recognized as allowed) .

In the Plan-recognition based model, CognitiveCentre discovers norms with a modified version of the "Norm detection algorithm" shown in Subsection 1.3. Having a look again at the pseudo-code it is clear that we need a planning and a plan-recognition algorithm. Figure 4 shows the architectural diagram of the Plan-recognition based prototype.

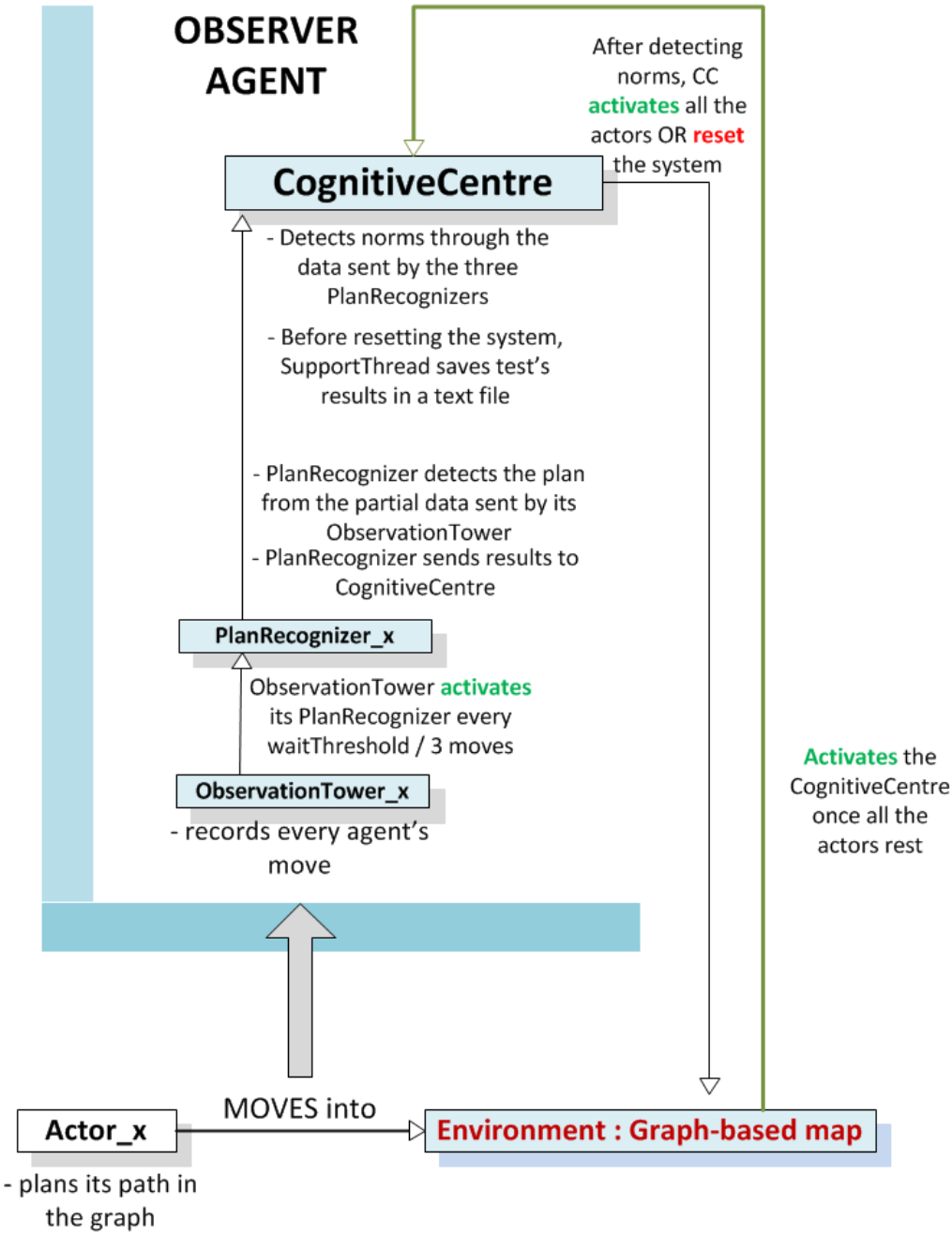


Figure 4: Plan-Recognition based prototype

Plan-recognition is performed by 3 *PlanRecognizer* threads. We have one PlanRecognizer for each ObservationTower, and so one for each actor. PlanRecognizers, as for ObservationTowers, run in parallel to actors. However a PlanRecognizer, instead of its ObservationTower, can record only some of the moves performed by the observed actor. For this purpose we use a variable parameter, called *waitThreshold*. A PlanRecognizer records the actor's node position every waitThreshold/3 moves. So, for example, if waitThreshold is equal to 3, PlanRecognizer records actor's position every move; this means that PlanRecognizer makes no mistakes[9] and we are in the *base case*.

The PlanRecognizer threads recognize actors' plans by making use of a modified version of the Symbolic Plan Recognizer (SPR), described in Subsection 1.1. Note that the changes made to the original code are due to the different, and more complex, system described in this work. First of all we have to translate the graph into a plan-library as seen in the Figure 1 of the "Symbolic Plan Recognizer", but this means storing in the PlanRecognizer a huge and redundant structure which specifies all the possible plans (with every possible destinations) for every starting node. Therefore we preferred to create a partial structure dynamically, depending on the agent's initial position and on the number of its moves. The plan library implemented in our code is made of *PlanLibNode* objects. PlanLibNode intuitively models a SPR plan library node. Following the terminology of the SPR work, every PlanLibNode have a reference to a vertical parent (another PlanLibNode), a reference to a sequential parent (idem), a reference to a list of vertical children (i.e. an arraylist of PlanLibNodes) and a reference to a list of sequential children (idem). Every plan library structure stored in the PlanRecognizer has one root, which is a PlanLibNode without neither sequential nor vertical parents. PlanLibNodes also contain a *name* and a *tag* field.

When the system starts, a PlanRecognizer immediately calls the *initPlan* method. The procedure records the agent's initial position, and looks for all of its neighbours, namely the nodes connected with the starting one by an arc. The initial plan-library graph is created: the root is called with the number of the starting node, and the first vertical level (root's children) corresponds to the neighbours of the starting node. Then this partial graph is extended by 2 levels: we look for all the neighbours of the neighbours of the root, and so on. An example of an initial plan-library graph, made for an agent whose starting position is 0 in Figure 3, is shown in Figure 5. Note that the root PlanLibNode is called "0" and after the first level (populated by PlanLibNodes 1 and 15), only sequential edges are used: this method partially differs from the original structure but doesn't affect the algorithm's final result.

[9] the system becomes fully observable

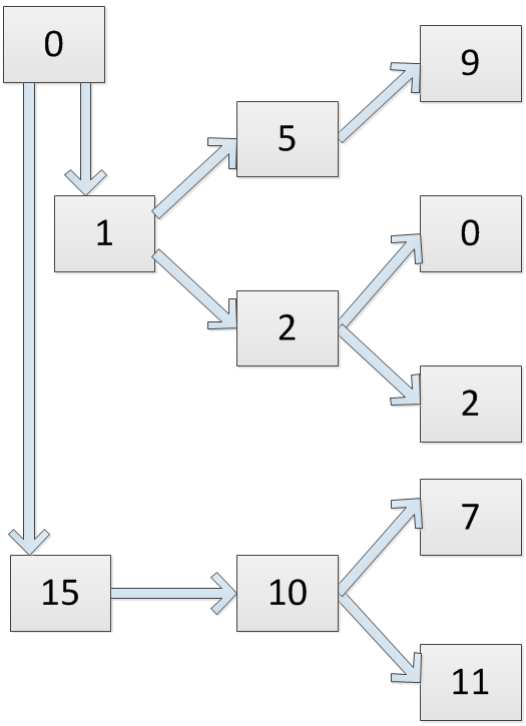


Figure 5: an example of our plan-library

With the completion of initPlan, PlanRecognizer waits (on a semaphore). ObservationTower has the duty to reactivate it every waitThreshold/3 or if the actor reaches its final destination. When PlanRecognizer wakes up, it checks the number of moves performed by the actor, which is provided by the ObservationTower, and records the agent's current position (its position when ObservationTower activated it). At this point if the number of moves is greater than or equal to plan-library levels, *extendPlan* is called. This method simply extends the library by 3 levels. Having done all of this, the thread scans the list of the PlanLibNodes at level l, where l is equal to the recorded number of moves[10]: if one or more nodes are named with the same number as the current position, it is tagged with t. A value t is equal to 1 at the first iteration and is incremented by one every time ObservationTower activates its PlanRecognizer. The tag in the system intuitively represents the temporal information as shown in [11].

If t is 1, then possible plans are built by finding (and reversing) all the paths from the tagged nodes to the root. Otherwise, the method *propagateUp* is called, which checks temporal consistency in a similar manner to that described

[10] Note that if we couldn't use the "recorded number of moves" information, the number of possible plans would have been much larger: this simplification reduces the difficulty in adapting the SPR approach to our environment.

in Section 1.1: propagateUp traverses the tree-like structure from every tagged node; when it finds a tagged ancestor of that node checks its tag. If the tag is t - 1, then the current node is temporally consistent, so the algorithm will simply select the next tagged node of level l. If the tag is less than t - 1, or there are no more tagged nodes up until the root, then the current node is temporally inconsistent: the tag is removed from it and the next tagged node of level l is selected. As previously seen, new possible plans can be detected by finding all the paths from tagged nodes to root node.

It is important to note one of the main differences from the original Symbolic Plan Recognizer algorithm: now a node is *temporally consistent* if one of these two conditions holds:

- it is tagged with "1"; or
- it is tagged with $t > 1$ and there exists an ancestor which has been tagged with *t - 1*

This different definition is due to the fact that there is no full observability of the actors' moves: instead of what we have described in Section 1.1, the observer cannot record the agent's moves at every step. Therefore if a node is tagged with t, it is not necessary that its sequential parent has the previous tag. However we have to be sure that the current tagged node is part of a started plan, which means that the PlanRecognizer has recorded, at *t-1*, another node which is part of the same plan. Note also that here there is no real distinction between sequential and vertical edges. Vertical edges connects the root to its neighbours and sequential edges are used for everything else. This use is not faithful to the original work but does not affect the final results. If a node is not temporally consistent, the tag is removed from it. So neither the "pruned" node will be used to build the possible recognized plans nor its ancestors unless one, or more, of these are part of temporally consistent plans. Figure 6 should make the whole process clearer for the reader.

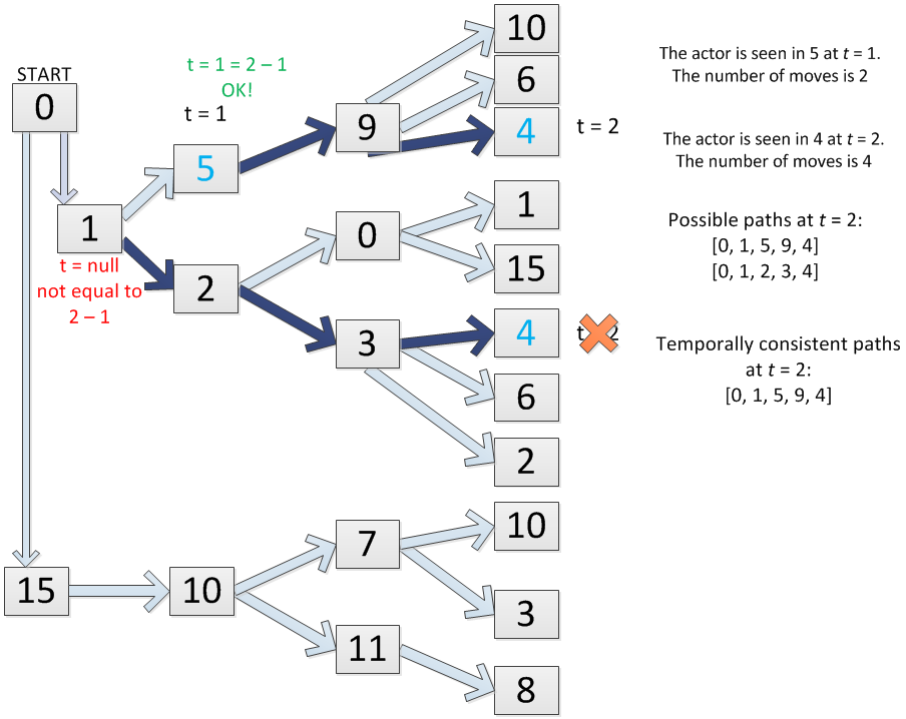


Figure 6: an example of the *temporal consistency* checking

In the example waitThreshold is equal to 6. Initially ObservationTower finds its followed actor in position 0 (of the graph shown in Figure 3) and the partial plan-library is created by the Plan-Recognizer. After 2 moves, PlanRecognizer wakes up, for the first time, the actor is seen to be in 5. The 2nd level of the graph is populated by these PlnaLibNodes: 5, 2 and 10. Node 5 is tagged with 1 and the only possible path is [0, 1, 5]. Then, after 4 moves, the PlanRecognizer wakes up again and finds (through its ObservationTower) the actor in 4: there are two nodes named as "4" at level 4 of the library and they are both tagged with 2. The value of t this time is greater than 1, so PropagateUp procedure is called for the two tagged nodes. The path [0, 1, 5, 9, 4] is temporally consistent because there is a PlanLibNode tagged with 2 - 1 (Node 5). The other possible path ([0, 1, 2, 3, 4]) is not temporally consistent because from Node 3 to the root 0 there isn't any tagged node. So this path will be discarded by the algorithm, namely PropagateUp will remove the tag 2 from the PlanLibNode 4 which is child of PlanLibNode 3.

At the end of the consistency check procedure, all of the possible plans are saved[11] for the CognitiveCentre and then PlanRecognizer becomes inactive (waits on a semaphore) again.

[11] note that they could change in the next reactivation of the PlanRecognizer

The Cognitive Centre

CognitiveCentre thread wakes up immediately after all of the actors have called the rest procedure. Once active, CognitiveCentre uses the data stored by PlanRecognizer threads in a similar way to what has been shown in Subsection 1.3. For every actor, the centre owns a list of ArrayLists which contain the possible paths (or plans) of the actors. Looking at the code shown in 1.3, the reader should notice that the algorithm requires only one recognized plan: considering that CognitiveCentre has no other useful information, it has to choose randomly one possible recognized plan for every actor. CognitiveCentre also needs its own planning system. Having chosen one plan for the agent, the thread selects the first node of the plan (the starting position) and the last node (the destination). Then it plans its own path using the A* algorithm, including the prohibited arcs.

Now, similar to the original algorithm, the planning system's path specifies all of the arcs which are potentially prohibited. The recognized path instead specifies all of the arcs which are not prohibited thanks to the assumption that no actor will ever violate norms. So we create the *possibleArcs* ArrayList and we insert into it the arcs from the planning system's path. Then we add to the *notProhibitedArcs* ArrayList, initialized at the creation of the Cognitive Centre, the arcs from the detected plans by PlanRecognizers. The *potentialProhibitions* ArrayList, which is created at the beginning as the *notProhibitedArcs* set, is updated by adding the arcs from *possibleArcs* and then by removing the *notProhibited* arcs from the resulting set. Note that *notProhibitedArcs* and *potentialProhibitions* are created once and then updated every time the CognitiveCentre is activated, but *possibleArcs* is created from scratch at every iteration of the algorithm.

After the update of the *potentialProhibitions*, CognitiveCentre send a signal to the 3 resting actors[12] so they can start again.

2.3.1 Time Complexity

We have so far not discussed the time complexity of our algorithms, but will provide a brief examination of this within the current section, evaluating the time complexity of the PlanRecognizer and CognitiveCentre's main methods.

At every iteration (i.e. every time PlanRecognizer is activated by its ObservationTower) PlanRecognizer scans all the PlanLibNodes at level l, where l is equal to the current number of moves. In order to compute the number of them, we must examine the structure of the Plan Library. The Plan Library can be represented as a tree-like structure, where the Root node stores agent's initial position in the graph and each tree node has as many children as the number of the neighbours of the graph's node stored in it (look at *initPlan* and *extendPlan* methods). If N is the number of graph's nodes, every node has at most $O(N)$ neighbours. Thus we can claim that if the Plan Library has l levels, the number of its nodes is $\boldsymbol{O(N^l)}$. Now we have to evaluate the *work* which

[12] sending a signal to their semaphores

is done in each node of the graph. Two methods are identified as potentially time-demanding: *propagateUp* and *adjacencyNode*, which is the one that locates a node's neighbours. The *adjacencyNode* method's current implementation is somewhat naive: it scans all the graph's arcs in order to find one or more which connect the selected node to another one. If A is the number of arcs in the graph, *adjacencyNode*'s time complexity is equal to A.

The *propagateUp* method traverses the graph from a tagged node (at level equal to l) up until the root in the worst case. This path, which is repeated by each tagged node, is $O(l)$. If t is the number of tagged nodes at level l, *propagateUp*'s time complexity is $O(t\ l)$. Thanks to a significant amount of tests we can claim that the number of tagged nodes for which *propagateUp* reach the root is much smaller than the number of the graph's nodes or arcs and can be treated as a constant. So *propagateUp*'s time complexity is simply $O(l)$ with this approximation. Overall the run-time complexity of PlanRecognizer is $\boldsymbol{O(N^l(A + l\ ))}$. It is important to note that the maximum value of l is more than or equal to the length of the final agent's path detected by the observer. This one is much smaller than the size of the graph in the average case but in the worst case it is $O(min\{N,A\})$, or, in other terms, if the graph size is denoted with G, it is $O(G)$. Therefore the worst case time complexity of PlanRecognizer is $\boldsymbol{O(G^G)}$, which is extremely inefficient. Luckily, at least in the model described in this work, the worst case is also the rarest one[13].

CognitiveCentre simply works with a list of paths and plans the optimal path using the A* algorithm. In particular CognitiveCentre calls A* with heuristic function equal to zero. Considering that the arc's cost is equal to one, A* behaves like a breadth-first search algorithm on a graph with time complexity $O(\ N + A\ )$. As for the remaining work, referred to the lists' editing, it can be easily shown that its time complexity is $O(min\{N,A\}^2)$. Overall the run-time complexity of CognitiveCentre is $\boldsymbol{O((N+A)^c)}$, where c is a constant.

[13] In all our experiments, this bound was never encountered

2.4 Violation based prototype

In the Violation-based prototype, the environment and the actors are modified in order to recreate a similar system to the one described in [16] (see subsection 1.2). Figure 7 shows the architectural diagram of the Violation based prototype.

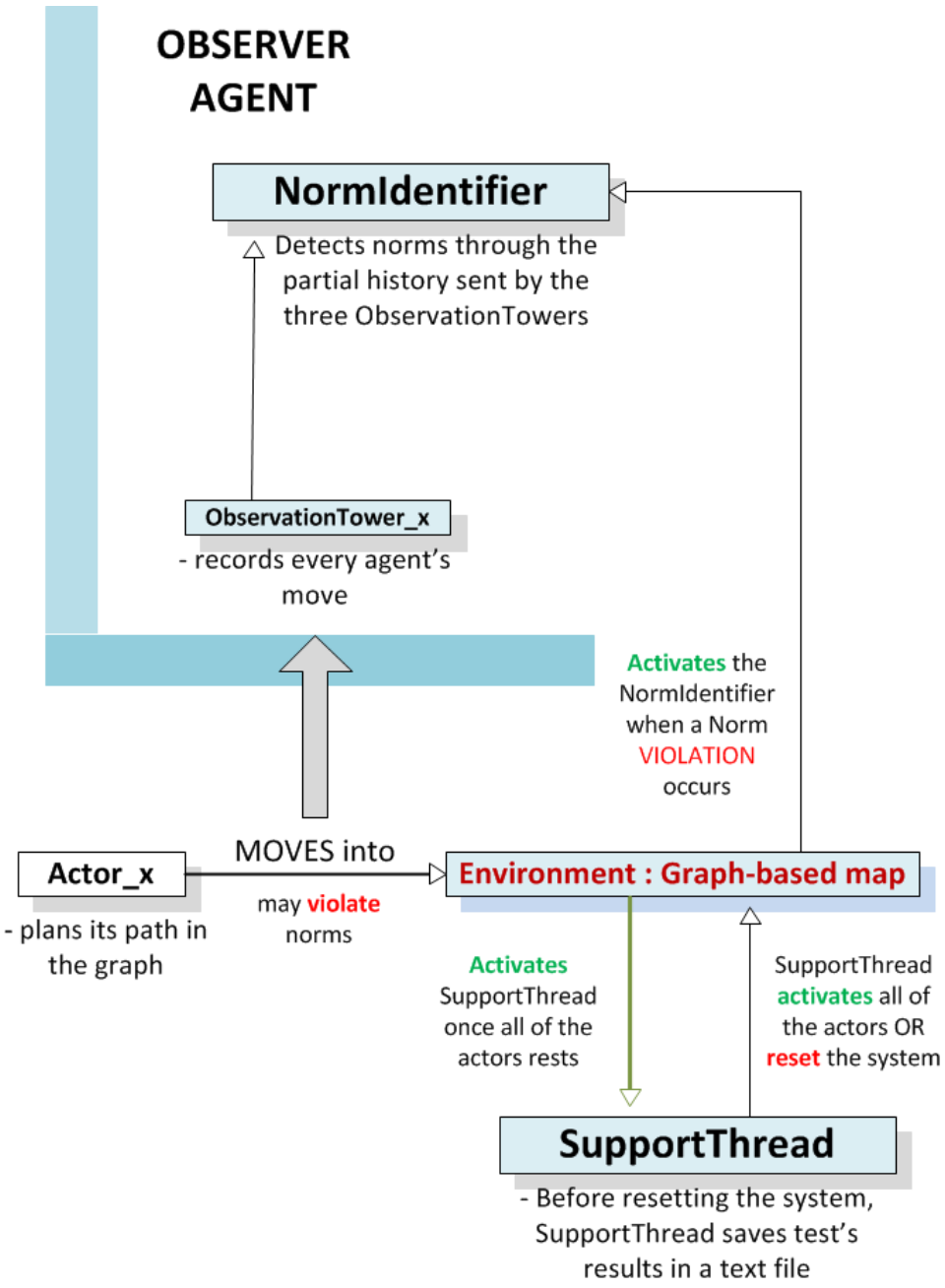


Figure 7: Violation based prototype

Instead of what has been seen in 2.3, moving agents are supposed to sometime violate norms and a violation system has to make the observer aware of these events. When an actor plans the path towards its destination, the *randPlan* procedure includes prohibited arcs[14] in the final path with probability $1/2$ or simply plans as seen before with probability $1/2$. When an actor goes through a prohibited arc, a *violation-event* occurs. This activates the *NormIdentifier* thread that has the purpose of deducing norms starting from the last and the previous violations. After doing that, NormIdentifier waits on a semaphore for the next violation.

As seen in the Plan Recognizer based prototype, when an actor reaches its destination, it calls the rest procedure. When all of the 3 agents are resting, *SupportThread* is activated in order to send a signal to them so they can restart. Note that, in the previous subsection, CognitiveCentre had the same job as is done here by NormIdentifier (norm recognition) and by SupportThread (system reset). Note also that even if the probability for an agent of making a plan which ignores prohibited arcs is $1/2$, the probability of making a plan which INCLUDES one or more prohibited arcs is different (less). In fact a prohibited arc could be discarded in the plan because it is not convenient.

The NormIdentifier

The NormIdentifier implements the algorithm seen in the Section 1.2. Here a prohibition is only a single arc, so there aren't, in our environment, forbidden sequences of arcs[15]. For this reason NormIdentifier is inspired by algorithm 1,2 and 3 of the *Candidate Norm Inference* procedure: that is the only basic approach to recognize norms through an agent's single-event violation. When a prohibited arc is crossed, the violation activates the NormIdentifier. Similarly to what is done in [16], the identifier thread doesn't have full observability of the agent's moves: in this case, when NormIdentifier is activated, it only knows that a violation has occurred recently, but it can't know a-priori which agent is responsible.

Once activated, NormIndentifier scans all the (previous or simultaneous to the violation) moves' history: every detected arc is put on a list (HashTable) and associated with the number of times that it has been used. Then if the scanned arc belongs to the last WS[16] visited arcs of a particular actor, where WS is here called *windowSizeHistory*, it is inserted in the *prohArcs* ArrayList. After this first phase, pruning work is performed. NormIdentifier checks the number associated with every arc in *prohArcs*: if that number divided by the number of total moves is greater than or equal to *prohibitedArcThreshold*, which corresponds to the Norm Pruning Threshold seen in Algorithm 5, the current arc is removed from the *prohArcs* list. The resulting ArrayList contains all of the potential single-arc prohibitions. The NormIdentifier, at this point, waits on a semaphore for the next violation.

[14] it ignores norms

[15] This could be a good starting point for future work.

[16] see subsection 1.2

2.4.1 Time Complexity

NormIdentifier's time complexity is simple to evaluate and it is directly connected with the *insert* and *find* methods in a (Java) HashTable and an ArrayList. All of them run in linear time in the worst case. Due to at least one loop in the code, it can be easily claimed that the worst case run-time complexity of NormIdentifier is $\boldsymbol{O(A^2)}$, where A is the number of the graph's arcs.

3 Test result

In this section we will show the effectiveness of these two approaches in different cases. Note that the two approaches utilize different assumption (for example in the first one, actors must not violate norms, in the second one, actors are sometime expected to violate norms). Future works could expand our prototype through additional tests to provide a more realistic comparison between approaches. For now we will establish if and in which partial-observability conditions, (symbolic) plan-recognition could be useful to infer norms under ideal conditions and if it could beat a basic violation-based norm-identification working under ideal assumptions.

Every test is made up of 40 iterations. In each iteration, the system generates 5 prohibited arcs and runs until the CognitiveCentre or SupportThread finds out that the total number of moves is greater than or equal to 400. This number has been chosen because in early tests with the plan-recognition based prototype in its base case, the CognitiveCentre nearly always took less than 400 moves to correctly identify all of the prohibited arcs.

3.1 Plan-recognition based prototype

3.1.1 Base case: waitThreshold = 3

Here we will show the result of the plan-recognition based prototype with waitThreshold equal to 3. This means that PlanRecognizer threads wakes up after every move of the observed actor. The reader should notice that in this particular situation, plan-recognition is pointless because it is equivalent to an observing entity which records the whole actor's path. We can deduce a-priori that in the base case there will never be false negatives. A false negative is a prohibited arc which is recognized as a permitted one. This will never happen here because if all of the used and allowed arcs (plan-steps) are successfully recognized, only them will be inserted in the *notProhibitedArcs* set. However if an arc is never used by actors, it could be represented as a false positive by being inserted in the *potentialProhibitions* set.

Results are shown below in the Table 1. The "Recognized arcs" column is composed by suming the "Successfully rec." column with "false negative". "Mistakes" is the result of the sum of the "false positive" column with "false negative".

Test n.	Recognized arcs	Successfully rec.	False Positives	False Negatives	Mistakes
1	5	5	0	0	0
2	3	3	0	0	0
3	5	5	1	0	1
4	5	5	0	0	0
5	5	5	0	0	0
6	5	5	0	0	0
7	5	5	0	0	0
8	5	5	0	0	0
9	4	4	0	0	0
10	5	5	0	0	0
11	5	5	0	0	0
12	5	5	0	0	0
13	5	5	0	0	0
14	5	5	0	0	0
15	5	5	0	0	0
16	5	5	1	0	1
17	5	5	0	0	0
18	5	5	0	0	0
19	5	5	0	0	0
20	5	5	0	0	0
21	5	5	0	0	0
22	5	5	2	0	2
23	5	5	0	0	0
24	5	5	0	0	0
25	5	5	0	0	0
26	4	4	0	0	0
27	5	5	0	0	0
28	5	5	0	0	0
29	5	5	0	0	0
30	5	5	0	0	0
31	3	3	1	0	1
32	5	5	1	0	1
33	4	4	0	0	0
34	4	4	0	0	0
35	3	3	0	0	0
36	5	5	0	0	0
37	5	5	0	0	0
38	5	5	0	0	0
39	5	5	0	0	0
40	5	5	0	0	0
AVR	**4,75**	**4,75**	**0,15**	**0**	**0,15**

TABLE 1: waitThreshold = 3

As seen in the table the number of successfully recognized is almost perfect (5) at every iteration. Moreover the number of false positives is low, 0 on average and not more than 2. As expected there are no false negatives.

An interesting fact is that the false positives are almost always the same arcs. They underline a phenomenon which could have been intuitively predicted but which was not mentioned in [18]. Arcs which are not convenient for the (experienced) actors in a system will likely never detected as allowed by a new agent in the system unless it tries the unused arc.

3.1.2 waitThreshold = 6

PlanRecognizer threads now wake up after every 2 moves of their observed actors. This case is the first one where the PlanRecognizer algorithm is really useful and the first one, at the same time, which points out the real flaw of our basic plan-recognizer approach. Let us show this with a simple example referring to Figure 3. The system generates (9, 13) as a prohibited arc, and an actor's starting position is 9. This actor wants to go in 11, so it chooses this path: [9, 10, 11] using the allowed arcs (9, 10) and then (10, 11). Considering that its PlanRecognizer wakes up every 2 moves, the actor is seen in 9 as initial position and, after 2 moves, it is seen in 11. Now from the observer point of view two paths are possible and temporally consistent: [9, 10, 11] and [9, 13, 11]. The choice between these two possible plans is random. If the second path is chosen, the NormIdentifier will put, in the notProhibitedArcs set, the arc (13, 11), which is allowed, and the arc (9, 13) which is forbidden. This generates a false negative. Notice that the NormIdentifier algorithm behaves in a greedy way with the notProhibitedArcs set: once an arc is put into this set, it will be never removed from it. This suggests a flaw of the Norm-Identification approach which will be examined later. Results of this test are shown in the Table 2.

Test n.	Recognized arcs	Successfully rec.	False Positives	False Negatives	Mistakes
1	4	2	0	2	2
2	4	4	0	0	0
3	5	5	1	0	1
4	3	3	0	0	0
5	5	5	1	0	1
6	3	3	1	0	1
7	3	3	2	0	2
8	4	4	1	0	1
9	5	5	1	0	1
10	4	4	0	0	0
11	2	2	0	0	0
12	4	3	1	1	2
13	6	5	0	1	1
14	5	5	0	0	0
15	5	4	0	1	1
16	5	4	2	1	3
17	5	5	1	0	1
18	5	5	0	0	0
19	5	1	1	4	5
20	5	5	2	0	2
21	3	2	0	1	1
22	3	3	0	0	0
23	5	4	0	1	1
24	5	4	0	1	1
25	5	5	0	0	0
26	5	4	0	1	1
27	4	3	1	1	2
28	5	3	0	2	2
29	5	5	1	0	1
30	5	4	1	1	2
31	5	3	0	2	2
32	4	4	1	0	1
33	5	3	0	2	2
34	4	2	1	2	3
35	3	3	0	0	0
36	5	4	0	1	1
37	5	3	0	2	2
38	5	4	0	1	1
39	5	3	0	2	2
40	5	5	0	0	0
AVR	**4,45**	**3,7**	**0,48**	**0,75**	**1,23**

TABLE 2: waitThreshold = 6

As expected, the number of recognized forbidden arcs is more or less the same as the previous situation. However some of them are now identified as allowed: we have some false negatives. Their number is still low as for the number of total mistakes at every iteration.

3.1.3 waitThreshold = 9

PlanRecognizer threads now wake up every 3 moves of their observed actors. We expect a higher number of false negatives for the reasons explained before. Results are shown in the Table 3.

Test n.	Recognized arcs	Successfully rec.	False Positives	False Negatives	Mistakes
1	4	2	0	2	2
2	5	3	0	2	2
3	5	1	1	4	5
4	5	2	0	3	3
5	5	2	0	3	3
6	5	0	0	5	5
7	5	3	0	2	2
8	5	2	0	3	3
9	5	1	1	4	5
10	5	0	0	5	5
11	5	2	0	3	3
12	5	2	0	3	3
13	5	2	0	3	3
14	5	3	0	2	2
15	5	2	0	3	3
16	5	1	0	4	4
17	5	1	0	4	4
18	5	2	0	3	3
19	5	1	1	4	5
20	5	1	1	4	5
21	4	0	0	4	4
22	5	0	0	5	5
23	5	2	1	3	4
24	5	0	0	5	5
25	5	2	0	3	3
26	5	2	1	3	4
27	5	0	1	5	6
28	5	3	0	2	2
29	5	3	0	2	2
30	4	0	0	4	4
31	5	0	0	5	5
32	5	1	1	4	5
33	5	2	0	3	3
34	5	2	0	3	3
35	5	0	0	5	5
36	5	2	0	3	3
37	5	2	0	3	3
38	5	1	1	4	5
39	5	0	0	5	5
40	5	1	0	4	4
AVR	**4,93**	**1,4**	**0,23**	**3,53**	**3,75**

TABLE 3: waitThreshold = 9

The number of recognized arc is still near 5 as before and the number of false positives is still low (lower than before). However the number of false positives heavily increased from the previous case and so did the number of total mistakes. Now they are 3.75 on average (11% of all the arcs). Moreover the number of successfully recognized arcs is now really low (1.4 on average against 3.7 previously). It is clear that our algorithm got worse by increasing the WT from 6 to 9. Looking deeply at the table we could deduce that the result, which mostly happens, is this: 2 successfully recognized arcs, 0 false positives and 3 false negatives.

3.1.4 waitThreshold = 12

PlanRecognizer threads now wake up every 4 moves of their observed actors, or at their last move before resting. Results are show in Table 4.

Test n.	Recognized arcs	Successfully rec.	False Positives	False Negatives	Mistakes
1	5	0	0	5	5
2	5	0	0	5	5
3	5	1	0	4	4
4	4	1	0	3	3
5	5	0	0	5	5
6	4	0	0	4	4
7	5	0	0	5	5
8	4	0	0	4	4
9	5	0	0	5	5
10	5	0	0	5	5
11	5	3	0	2	2
12	4	0	0	4	4
13	5	1	0	4	4
14	5	2	0	3	3
15	4	1	0	3	3
16	5	0	0	5	5
17	5	0	0	5	5
18	5	2	0	3	3
19	5	1	0	4	4
20	5	0	0	5	5
21	5	1	2	4	6
22	5	1	1	4	5
23	5	1	0	4	4
24	4	0	1	4	5
25	5	0	0	5	5
26	5	0	0	5	5
27	5	1	0	4	4
28	5	0	0	5	5
29	5	0	1	5	6
30	5	1	0	4	4
31	5	1	0	4	4
32	5	0	1	5	6
33	5	0	1	5	6
34	5	2	0	3	3
35	5	0	0	5	5
36	5	1	0	4	4
37	5	0	0	5	5
38	5	1	1	4	5
39	5	0	0	5	5
40	5	1	0	4	4
AVR	**4,85**	**0,58**	**0,2**	**4,28**	**4,48**

TABLE 4: waitThreshold = 12

Performance is now very poor. The number of false positives is low, but the number of false negatives is almost 5 and, consequently, the number of successfully recognized arcs is mostly 0 or 1. The "average" result, which happens for most times, is 5 false negatives, 0 false positives, 0 successfully recognized arcs. We can say that we reached the first worst-performance (at least looking at the number of false negatives) case and therefore do not increase waitThreshold further.

3.2 Violation based prototype

Using this approach, we can vary 2 parameters in our tests. *windowSizeHistory* represents the number of moves before a violation-event which are examined by the NormIdentifier. We identify the base case as windowSizeHistory equal to 2: this case simulate the situation where the observer wakes up immediately after a violation and so can take into consideration only the last move (or 2 moves) of every agent. We saw that with WS equal to 1, the observer's number of (successfully or not) recognized prohibited arcs was low (1 or 2). This is due to the fact that the observer's awakening is not always coincident with the last violation.

prohibitedArcThreshold is a value which establishes if the arc examined has to be detected as potentially forbidden or not. We encountered difficulties in deciding which value could give better results, so we decided to start by setting it to 1 divided by number of total Arcs (35), i.e. 0.029.

3.2.1 Base case: windowSizeHistory = 2

In this case we will start by putting the prohibitedArcThreshold equal to 0.029 (in the original paper, the NormPruningThreshold was 1/2) and see how the algorithm behaves with these parameters. Results are shown in Table 5.

Test n.	Recognized arcs	Successfully rec.	False Positives	False Negatives	Mistakes
1	4	4	12	0	12
2	2	2	9	0	9
3	2	2	4	0	4
4	4	4	14	0	14
5	3	3	9	0	9
6	3	1	11	2	13
7	3	3	8	0	8
8	3	3	16	0	16
9	4	4	13	0	13
10	4	4	13	0	13
11	3	3	11	0	11
12	3	3	9	0	9
13	4	4	15	0	15
14	4	3	14	1	15
15	3	3	9	0	9
16	3	3	6	0	6
17	3	3	15	0	15
18	4	4	9	0	9
19	4	4	12	0	12
20	3	3	17	0	17
21	2	2	11	0	11
22	4	4	10	0	10
23	4	4	7	0	7
24	2	2	11	0	11
25	3	3	8	0	8
26	4	4	12	0	12
27	3	2	16	1	17
28	1	1	10	0	10
29	3	2	8	1	9
30	3	3	7	0	7
31	3	3	7	0	7
32	3	3	9	0	9
33	4	4	8	0	8
34	3	3	13	0	13
35	3	3	16	0	16
36	1	1	7	0	7
37	5	5	12	0	12
38	3	3	12	0	12
39	3	3	12	0	12
40	0	0	7	0	7
AVR	**3,08**	**2,95**	**10,73**	**0,13**	**10,85**

TABLE 5: windowSizeHistory = 2, prohibitedArcThreshold = 1 / 35

The number of recognized arcs is less than before, but all of them are correctly identified as prohibited. The real problem is that there are a high number of false positive. More generally the algorithm, with this prohibitedArcThreshold value, recognizes almost the 30 % of the arcs as prohibited. Connected with this, we have an average of almost 11 mistakes. To overcome this, the prohibitedArcThreshold parameter should be modified but we were unable to find an appropriate value. We can only deduce that the actual value is too high because the pruning procedure (see 2.4) takes place only a few times.

We decided to "help" the algorithm by varying prohibitedArcThreshold during execution: after pruning, the NormIdentifier thread checks if the size of the prohArcs set is more than or equal to 7. If it is, prohibitedArcThreshold is reduced by 0.01 (if it is more than or equal to 0.01). Instead, if prohArcs' size is less than 2, then the parameter is increased by 0.01. This modification is naive and actually helps our original algorithm because the observer shouldn't know a-priori the real number of the prohibited arcs. Note that in the original work [16] the NormPruningThreshold was fixed. The results of this trial are shown in the next table (Table 6).

Test n.	Recognized arcs	Successfully rec.	False Positives	False Negatives	Mistakes
1	4	1	3	3	6
2	4	2	1	2	3
3	4	0	3	4	7
4	4	2	4	2	6
5	4	1	1	3	4
6	4	0	5	4	9
7	5	2	4	3	7
8	2	0	4	2	6
9	3	1	6	2	8
10	4	2	3	2	5
11	4	3	0	1	1
12	4	2	3	2	5
13	2	1	5	1	6
14	5	2	4	3	7
15	4	2	2	2	4
16	3	1	1	2	3
17	5	2	2	3	5
18	5	2	2	3	5
19	4	2	0	2	2
20	4	3	4	1	5
21	3	2	1	1	2
22	5	1	1	4	5
23	4	3	0	1	1
24	5	4	2	1	3
25	5	3	4	2	6
26	5	4	1	1	2
27	3	0	4	3	7
28	2	1	3	1	4
29	4	0	2	4	6
30	4	1	1	3	4
31	3	3	2	0	2
32	5	1	0	4	4
33	5	4	1	1	2
34	5	3	2	2	4
35	3	1	1	2	3
36	4	3	1	1	2
37	4	0	5	4	9
38	4	0	3	4	7
39	4	2	1	2	3
40	4	3	2	1	3
AVR	**3,98**	**1,75**	**2,35**	**2,23**	**4,58**

TABLE 6: windowSizeHistory = 2, prohibitedArcThreshold = *variable*

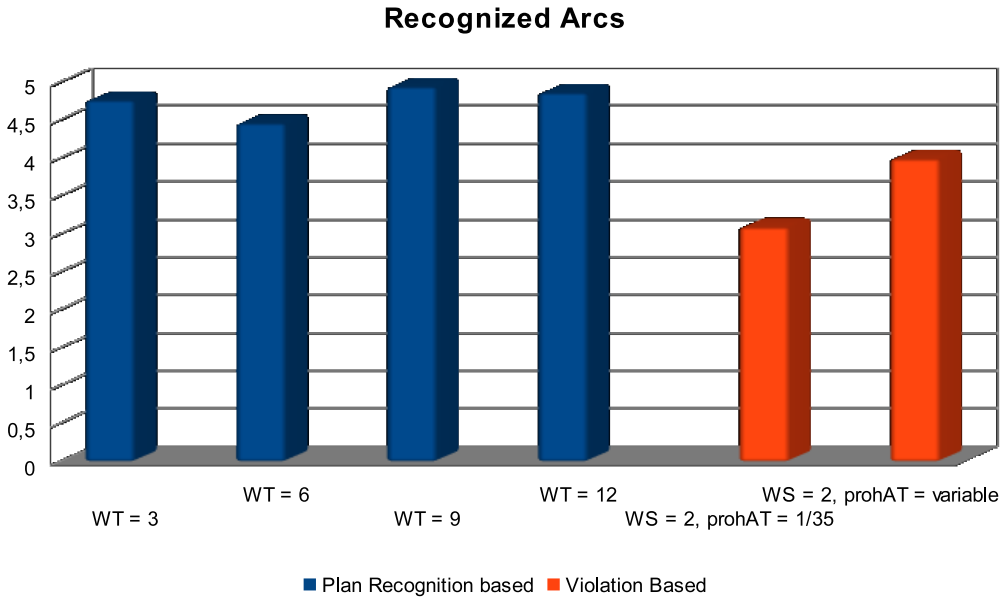
The modified algorithm now decreases the number of false positives, however the number of false negatives is higher than before, and higher, on average, than the number of successfully recognized arcs. More generally the results, and the number of total mistakes, are now unpredictable.

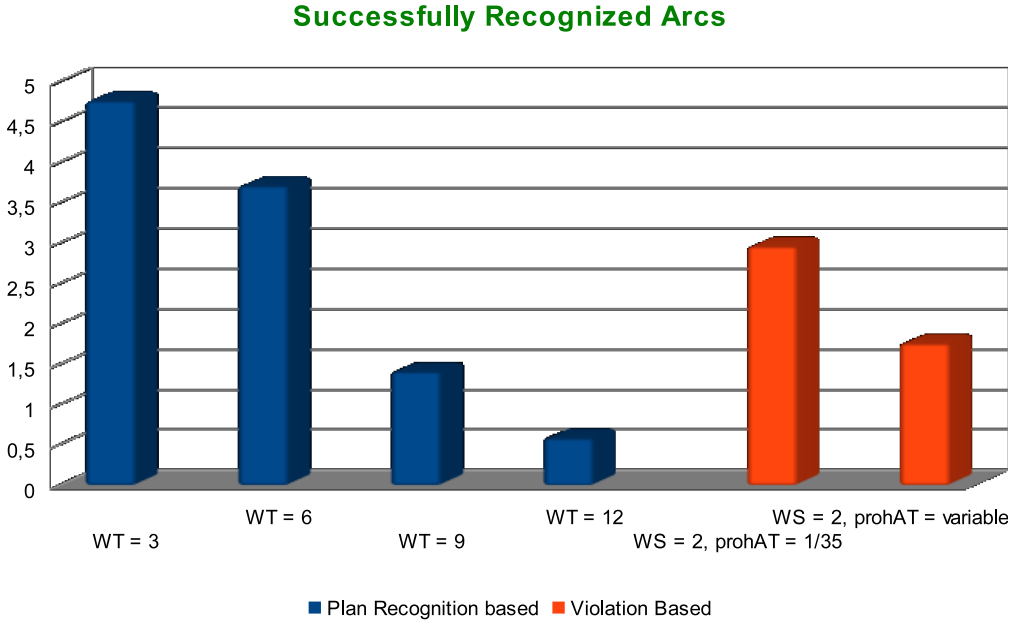
It is pointless to increase the windowSizeHistory because this would result in additional false positives, but the number of false negatives and of successfully recognized arcs would essentially remain the same. Note that the number of recognized arcs is already high in the base case.

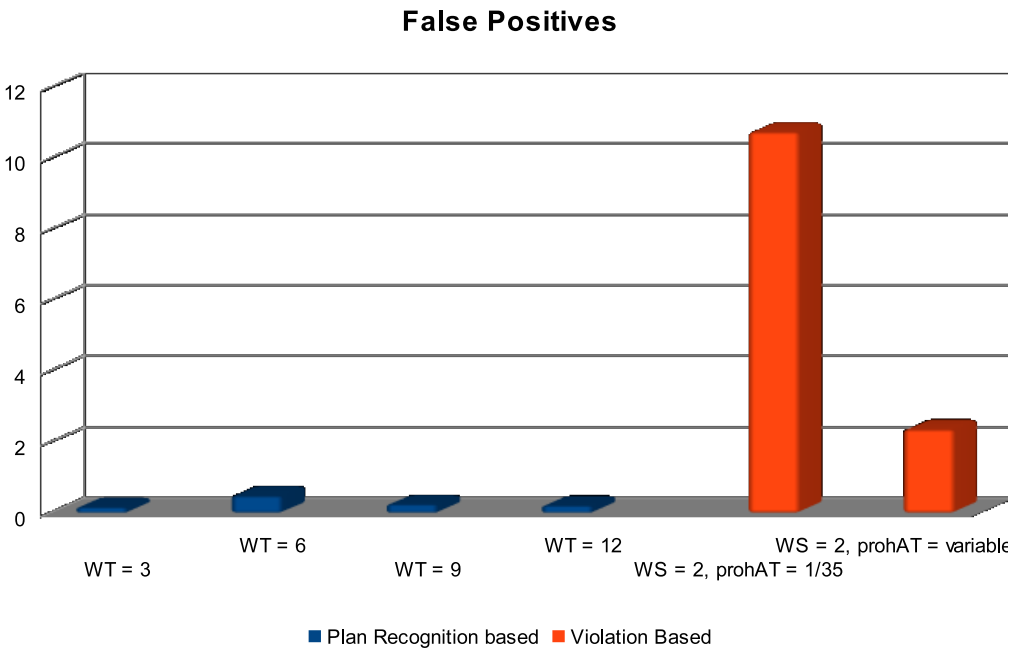
4 Conclusions and future works

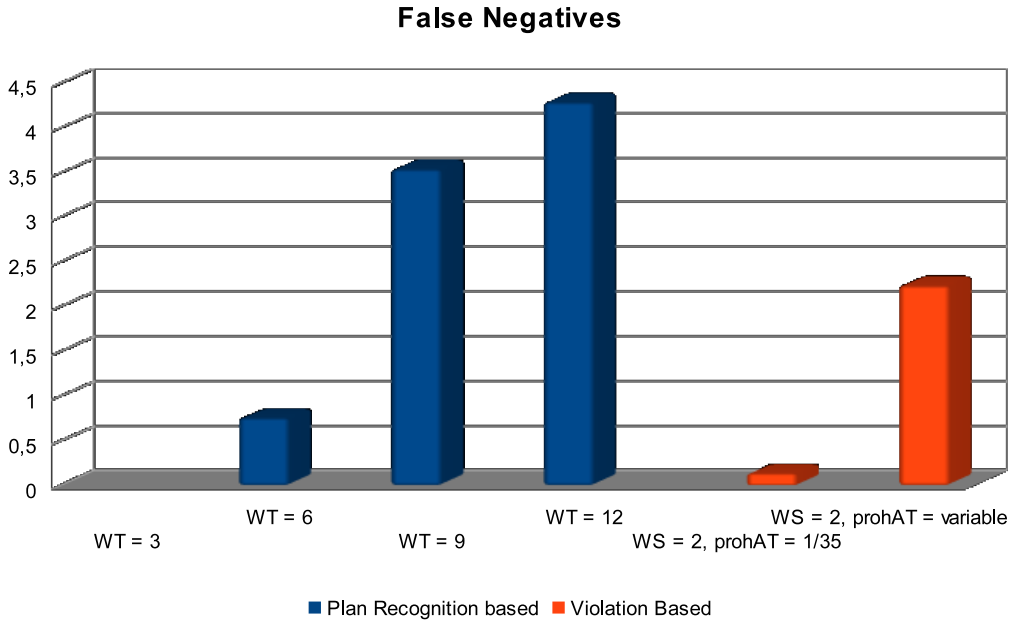
With our project we wanted to evaluate a norm-identification approach based on a basic (symbolic) plan-recognizer and to compare it with another observational approach, namely a violation-based norm-identifier. One interest in our work is to look at the size of the plan library which is definitely bigger than most of all the project presented before ours.

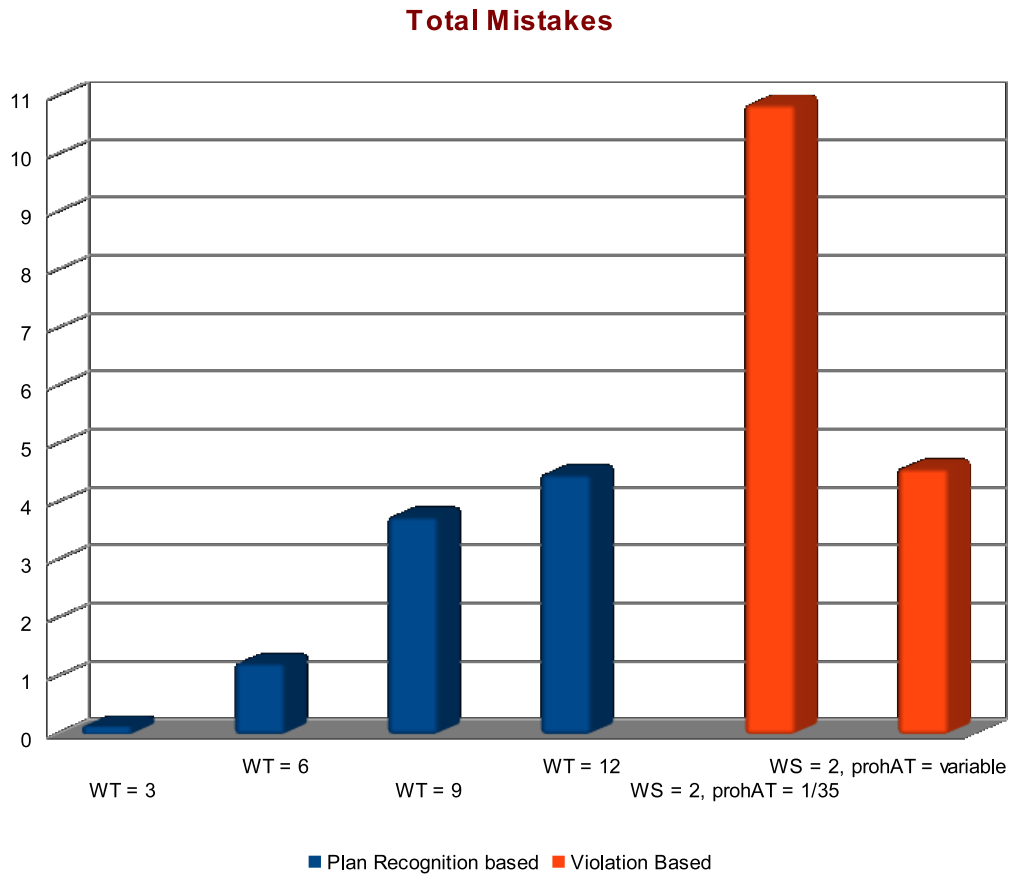
We summarize the results shown in the last section in the next series of graphs. Every column reports average values.


Recognized Arcs
5
4,5
4
3,5
3
2,5
2
1,5
1
0,5
0
WT = 3
WT = 6
WT = 9
WT = 12
WS = 2, prohAT = 1/35
WS = 2, prohAT = variable
Plan Recognition based
Violation Based


Successfully Recognized Arcs
5
4,5
4
3,5
3
2,5
2
1,5
1
0,5
0
WT = 3
WT = 6
WT = 9
WT = 12
WS = 2, prohAT = 1/35
WS = 2, prohAT = variable
Plan Recognition based
Violation Based


False Positives
12
10
8
6
4
2
0
WT = 3
WT = 6
WT = 9
WT = 12
WS = 2, prohAT = 1/35
WS = 2, prohAT = variable
Plan Recognition based
Violation Based


False Negatives
4,5
4
3,5
3
2,5
2
1,5
1
0,5
0
WT = 3
WT = 6
WT = 9
WT = 12
WS = 2, prohAT = 1/35
WS = 2, prohAT = variable
Plan Recognition based
Violation Based


Total Mistakes
11
10
9
8
7
6
5
4
3
2
1
0
WT = 3
WT = 6
WT = 9
WT = 12
WS = 2, prohAT = 1/35
WS = 2, prohAT = variable
Plan Recognition based
Violation Based

The violation-based norm-identifier seems to work well when the number of possible single events is low (2 or 3). In the original work the observer didn't have full observability of the environment, however the actor could only *move*, *eat* and *litter* in the system. In our system this approach performs poorly even in its base case and even if the observer is provided with the estimated number of the prohibited arcs via dynamically adapting the prohibitedArcThreshold parameter. Intuitively if an arc is used with a low frequency, it is not necessary prohibited, but could just not be convenient for all of the agents in the environment. On the other hand if an arc has medium-high frequency, the arc could be prohibited but convenient for the actors. The problem is that the observer has no information about how to choose a good value of *prohibitedArcThreshold*. Future work, maybe involving multi-arc norms (for example: prohibited sequences of allowed arcs), should avoid any use of this approach, at least if used alone, because of its unsatisfactory performance with single-arc norms.

Regarding the plan-recognizer based prototype, it works very well in its base case and still behaves fine with the waitThreshold equal to 6. However, if we increase the parameter to 9, its performance starts to degrade. The number of false positives is always low, but if we set waitThreshold equal to 12, the number of false negatives is almost 5. This norm-identifier algorithm therefore appears good but is still immature for a partially observable environment. Its main flaw is that it relies completely on the plan-recognition component: if a prohibited arc appears at least once in a recognized path, after a plan-recognizer mistake, it will be inserted in the notProhib set and it will never be removed from it. Overcoming this issue is a topic for future works, allowing the agents to operate in a more general environment where norms can be violated. Moreover the symbolic plan recognizer should be extended: a ranking system (maybe based on a probabilistic approach) between the temporary consistent paths is needed in order to avoid a random choice between them.

Our work suggests that Plan-recognition can be used for norm-identification via plan libraries in partially observable environments, and is also a better choice than other observational approaches. Graph-based environments like ours are a simple but challenging test-bed for any future systems. Finally note that our directed graph is not provided with different costs for its arcs and that there are not any forms of sanction after a norm violation. The introduction of these features can be a future extension of our environment.

5 Difficulties Encountered

In this last Section we will explain all of the main difficulties which have been found during the experience reported.

The author was not able to find in literature a precise and common definition of what a "*norm*" is in multi-agent systems. That's why we decided to give our own definition (in the Introduction) of *norms* trying to make it compatible with the other definitions found in the works read.

It has been necessary to read many texts about Plan Recognition in order to write the Background Section. Most of them show procedures or algorithms which didn't fit with the environment described in Subsection 2.1 or simply were hard to implement in the languages known by the author. Moreover we didn't know how to add an utility function to the observer and to the actors in our environment, so we decided to focus on the basic SPR and to leave its extension, shown in [12], for future works.

During the code implementation, it has been difficult to make all the Threads (PlanRecognizers, ObservationTowers, etc..) temporally concurrent and to make them work in a satisfactory way. Thus we decided to partially interleave Threads' execution as it is explained in Subsection 2.2. We strongly believe that this simplification doesn't go against our final goal (the evaluation of Norm-Learning approaches).

Our program makes use of concurrent programming and, in particular, of Semaphores for Thread's synchronization. As consequence of this lots of (synchronization) bugs have been found during the coding and much time has been spent fixing them.

Finally we didn't encounter any particular problems in running the described tests with the final version of the software.

6 Appendix A: User Manual

This section explains how to run the program described in Section 3. The final package includes the running jar, **streetModel.jar**, and its configuration file, **streetModel.CFG**. We will first explain how to run properly the provided software and, in Section 6, we will show the reader how to install it properly.

6.1 The Configuration File

Before running streetModel.jar, the configuration file has to be modified in order to conduct tests in the desired way. The configuration file, **streetModel.CFG,** can be opened by a text editor: Figure 8 shows how it looks like (if opened by Microsoft NotePad). The test described in Subsection 3.1.2 has been done by setting the configuration file as it is shown in Figure 8.

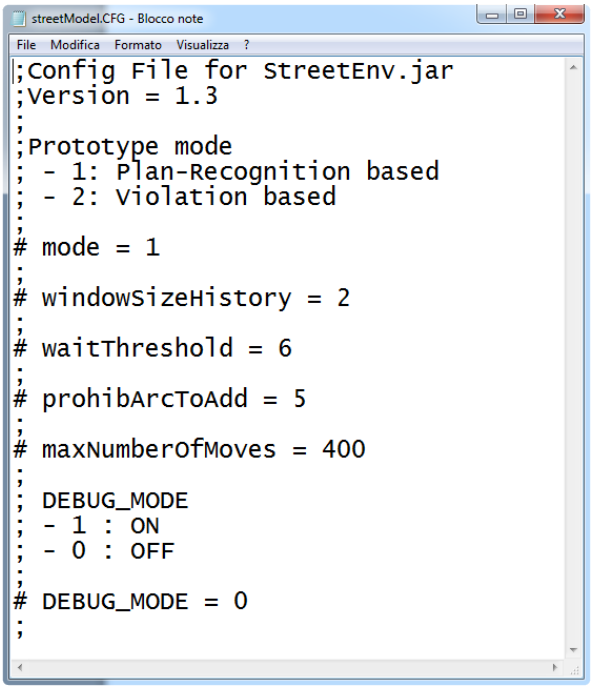


Figure 8: the "streetModel.CFG" configuration file

The user is expected to modify only the integer values in the lines starting with "#". Below is the description of all the parameters in the settings file.

mode: It can be "1" or "2". If it is 1, streetModel.jar will execute the Plan-recognition based prototype described in Subsection 2.3. Otherwise, street-

Model.jar will execute the Violation based prototype described in Subsection 2.4. Default value is equal to 1.

windowSizeHistory: This parameter has been described in Subsection 2.4 referring to the Violation based prototype. Default value is equal to 2.

waitThreshold: This parameter has been described in Subsection 2.3 referring to the Plan-recognition based prototype. Default value is equal to 6.

prohibArcToAdd: It indicates the number of prohibited arcs that are generated by the system at every iteration. Note that all our tests have been conducted by setting prohibArcToAdd equal to 5. Default value is equal to 5.

maxNumberOfMoves: CognitiveCentre (in the Plan-recognition prototype) or SupportThread (in the Violation based prototype) has to give the signal to actors, when they are resting, in order that they restart. Before doing that, it checks if the total number of moves is greater than or equal to the value of maxNumberOfMoves. If it is, the system is reset: new prohibited arcs take place of the old ones and all the observer's threads are recreated from scratch. Note that all our tests have been conducted by setting maxNumberOfMoves equal to 400. Default value is equal to 400.

DEBUG_MODE: It can be "0" or "1". If it is 1, more debugging messages will be shown in the program during its execution. The recommended value is 0. Default value is equal to 0.

6.2 The Jar File

After modifying properly the configuration file, the user can run the program by double-clicking streetModel.jar. The Jar file and the configuration file have to be in the same directory in order to work properly. An example of an execution of the software is shown below in Figure 9. The "Stop" button is used to quit the program and the "Pause" button is used to suspend the application. Once the "Pause" button is pressed, it will be replaced by the "Continue" button which can be used to resume the application. It is not recommended the use of the other buttons.

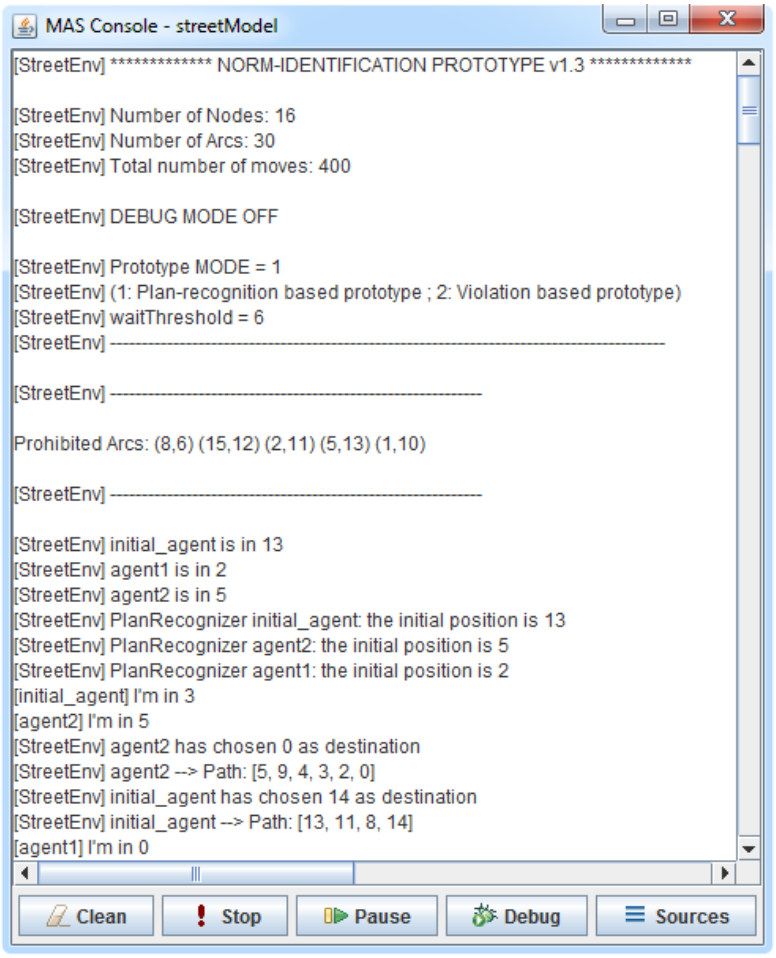


Figure 9: "streetModel.jar" running

If the application is not able find the configuration file, the program will display an error message and default values (see Section 6.1) will be used. Every time the program resets the system, it saves the results of the finished iteration in the file "Tests_Result_X.txt", where "X" is the iteration's number, if the Plan-Recognition based prototype is running. Otherwise it saves the results in the file "Tests_Result_Violation_X.txt".

7 Appendix B: Maintenance Manual

7.1 System requirements

The program requires a system with Java Runtime Environment 6, or later, installed to run. The JRE can be downloaded from

http://www.java.com/en/download/manual.jsp.

Our software has been tested only on a PC with Microsoft Windows XP. It is believed that the software should work also in other operating systems provided with JRE.

7.2 Installation

To install the program extract the 2 files (**streetModel.CFG** and **streetModel.jar**) from the archive **streetModel_Bin.zip**. After the JRE is installed, the program can be run by double-clicking on **streetModel.jar** or from a command line as follows:

```
java -jar streetModel.jar
```

7.3 Using streetModel

An user manual for the software is provided in the document 'User Manual' which is provided in Appendix A of the report.

7.4 Building streetModel

The source code is provided by the archive **streetModel_Src.zip**. Extract the directory "streetModel" from it. The project contained can be compiled using the modified version of jEdit available from http://sourceforge.net/projects/jason/. To run it in Windows, extract the directory "Jason-x.y.z" from the downloaded archive, where "x.y.z" is the number of the release version, and then open the file "jason.bat" which can be found in the directory "Jason-x.y.z\ bin". Once jEdit is running, open a saved project (File -> Open) and open the file "streetModel.mas2j" which can be found in the extracted directory "streetModel". To compile the project and make it a runnable Jar, go to "Plugin -> Jason -> Create an executable Jar". This will save an executable Jar of the project in the directory of the source code.

This is the only procedure to build the project. It is not possible for now to create a working Jar of a Jason/Java project with any other IDEs.

7.5 Running streetModel in Eclipse

Even if Eclipse IDE is not able to create an executable Jar of our project, it is possible to use the IDE to open the source code and to run it. To do that follow the instructions available from http://jason.sourceforge.net/mini-tutorial/eclipse-plugin/.

7.6 Source Code File List

The source code can be found in the directory "streetModel" provided by the archive **streetModel_Src.zip**. A full listing of files is included in Table 7.

File Name	Directory	Description
StreetEnv.java	streetModel\src\java	Implements the environment and the actors' actions as shown in Subsection 2.1 and 2.2
OrientedArc.java	streetModel\src\java	Defines the class OrientedArc which represents an oriented arc in the graph
ObservationTower.java	streetModel\src\java	Defines the Thread class ObservationTower as shown in Subsection 2.1 and 2.2
PlanLibNode.java	streetModel\src\java	Defines the class PlanLibNode which represents a plan library node as shown in Subsection 2.3
PlanRecognizer.java	streetModel\src\java	Defines the Thread class PlanRecognizer as shown in Subsection 2.3
CognitiveCentre.java	streetModel\src\java	Defines the Thread class CognitiveCentre as shown in Subsection 2.3
NormIdentifier.java	streetModel\src\java	Defines the Thread class NormIdentifier as shown in Subsection 2.4
SupportThread.java	streetModel\src\java	Defines the Thread class SupportThread as shown in Subsection 2.4
Config.java	streetModel\src\java	Contains a list of parameters whose default values are replaced by the values found in the configuration file streetModel.CFG
initial_agent.asl	streetModel\src\asl	Defines the plan of the first actor as shown in Subsection 2.1 and 2.2. All the actors are equivalent
agent1.asl	streetModel\src\asl	Defines the plan of the second actor as shown in Subsection 2.1 and 2.2
agent2.asl	streetModel\src\asl	Defines the plan of the third actor as shown in Subsection 2.1 and 2.2
streetModel.mas2j	streetModel	Defines the essential attributes of the project. See http://jason.sourceforge.net/mini-tutorial/getting-started/ for further details.

TABLE 7: Source Code File List

The files not included in the list are non-modifiable configuration files which are not relevant.

7.7 Known Bugs and Issues

The Jar file sometime stops to work unexpectedly when all of the actors in the system call the *rest* procedure (see Section 2 for further details).

7.8 Future improvements

Some Future improvements to the project and the provided application are included in the main report (Section 4). Here, simple operational improvements of the software will be proposed:

- Fix the bugs shown in Subsection 7.7.
- Improve the GUI of the application making it more user friendly: a graphical representation of the environment and of the actors' moves should be added.
- Make the user able to add or remove nodes and arcs to the environment.

8 Appendix C: Some Plan-Recognition Techniques

In this section we briefly describe some additional works on plan recognition which we considered implementing. They are not directly connected to the project as we settled on the SPR approach.

K. Myers [4] makes use of abductive reasoning in a intended case of intention recognition. The observer, a help/planning system, has to help the observed agent, the user, who has partial knowledge of the plan library, to reach his final goal. Every goal could be divided in sub-goals (necessary to reach the upper level goal) and every sub-goal could be divided into other sub-goals in a recursive way. The user can provide a goal or some subgoals to the help-system. In the first case the plan-recognition system is not involved but in the second the observer has to choose first one or more top-level plans applying abduction multiple times (abductive chain) starting from one subgoal. If the result of the abductive chain is a set of many top-level goals, a choice strategy is needed. However Myers doesn't suggest one. Myers' work has been reused recently by Jarvis and Lunt [5, 6] and applied to the adversarial case[17]. Here goals and subgoals become "actions" and "subactions", the subactions have to be executed in a particular order, the plan library is represented in the form of a template library and notions relating to the frequency of the action and accuracy of the observation are introduced. Before starting to apply the abductive chain, subactions are filtered by accuracy and frequency (the action is discarded if it has a high frequency or if its observation has a low accuracy) and then the same process, seen in the original Myers' work, takes place.

Mulder and Voorbraak [7] also used the abductive reasoning to "tactical"[18] intention recognition, which is de facto adversarial intention recognition applied to military examples. Given a set of observations the agent seeks a plan or a set of plans that includes the observed actions.

In recent work probability has been largely used as a form of global criteria to choose among the possible abductive results. Demolombe and Fernandez [8] assume full visibility of the actions of the observed agents, but in the environment an agent is allowed to reach a goal by interleaving actions which are necessary for the plan, called "explicit" actions, with other actions that are neither prohibited by the plan nor useful to it, called "tolerated" actions. Every goal is reached through an ordered list of explicit actions. After the matching of the first action with the observations' set, the probability of the plan/goal is increased if the next matched action corresponds to an expected explicit action. It is instead decreased if it corresponds to a tolerated action and decreased by a greater value if it corresponds to a prohibited action for that plan. At the end of this modified abductive process the plan and goal with greatest probability is chosen.

Charniak and Goldman [9] introduced a Bayesian network plan recognizer in 1993; this technique is still used and combined with the abductive reasoning. Pereira and Anh [10] represents the plan library as a Bayesian network of *causes*,

[17] terrorist intention recognition

[18] as called by the authors

intentions and *actions*. Every *cause* gives rise to one or more *intention* (with a certain probability) and every *intention* gives rise to one or more *action*. Every *action* has a pre-specified probability, every *intention* is exclusive and there is partial visibility of the environment. This class of recognizers was described and embedded in a keyhole type domain.

References

[1] Cohen, P.R., Perrault, C.R., Allen, J.F. *Beyond question answering.* In *Strategies for Natural Language Processing*, W. Lehnert and M. Ringle (Eds.), Lawrence Erlbaum Associates, Hillsdale, NJ, 1981.

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

[3] Fariba Sadri. *Logic-Based Approaches to Intention Recognition.* Department of Computing, Imperial College London, UK

[4] Myers, K.L. *Abductive completion of plan sketches.* In Proceedings of the 14 th National Conference on Artificial Intelligence, American Association of Artificial Intelligence (AAAI-97), Menlo Park, CA: AAAI Press, 1997.

[5] Jarvis, P., Lunt, T., Myers, K. (2004). *Identifying terrorist activity with AI plan recognition technology.* In the Sixteenth Innovative Applications of Artificial Intelligence Conference (IAAI 04), AAAI Press, 2004.

[6] Jarvis, P.A., Lunt, T.F., Myers, K.L. (2005). *Identifying terrorist activity with AI plan-recognition technology*, AI Magazine, Vol 26, No. 3, 2005.

[7] Mulder, F., Voorbraak, F. *A formal description of tactical plan recognition*, Information Fusion 4, 2003.

[8] Demolombe, R., Fernandez, A.M.O. (2006). *Intention recognition in the situation calculus and probability theory frameworks.* In Proceedings of Computational Logic in Multi-agent Systems (CLIMA) 2006.

[9] E. Charniak, R.P. Goldman. *A Bayesian model of plan recognition*, 1993.

[10] Pereira, L.M., Anh, H.T. *Elder care via intention recognition and evolution prospection*, in: S. Abreu, D. Seipel (eds.), Procs. 18th International Conference on Applications of Declarative Programming and Knowledge Management (INAP'09), Évora, Portugal, November 2009.

[11] Dorit Avrahami-Zilberbrand and Gal A. Kaminka. *Fast and Complete Symbolic Plan Recognition.* Computer Science Department Bar Ilan University, Israel 2005.

[12] Dorit Avrahami-Zilberbrand and Gal A. Kaminka. *Utility-Based Plan Recognition: An Extended Abstract.* The MAVERICK Group Computer Science Department Bar Ilan University, Israel 2007

[13] Guido Boella, Leendert van der Torre, and Harko Verhagen. *Introduction to the special issue on normative multiagent systems. Autonomous Agents and Multi-Agent Systems*, 17(1):1– 10, 2008.

[14] Mohamed Hamada. *Web-based environment for active computing learners.* In ICCSA (1), pages 516–529, 2008.

[15] Bastin Tony Roy Savarimuthu. *Norm Learning in Multi-agent Societies.* The Information Science Discussion Paper Series Number 2011/05, May 2011.

[16] Bastin Tony Roy Savarimuthu, Stephen Cranefield, Maryam A. Purvis and Martin K. Purvis. *Norm Identification in Multi-agent Societies.* The Information Science Discussion Paper Series Number 2010/03, February 2010.

[17] H. Mannila, H. Toivonen, A. Inkeri Verkamo. *Discovery of frequent episodes in event sequences*, *Data Mining and Knowledge Discovery 1* (3) (1997) 259–289.

[18] Nir Oren, Felipe Meneguzzi. *Norm Identification through Plan Detection (Extended Abstract).* Unpublished.

[19] Javier Vazquez-Salceda, Huib Aldewereld, and Frank Dignum. *Implementing Norms in Multiagent Systems.* Institute of Information and Computing Sciences Utrecht University, The Netherlands (2007)