

Incorporating Learning in BDI Agents

Stéphane Airiau
RMIT University
Melbourne, Australia
stephane@utulsa.edu

Lin Padgham
RMIT University
Melbourne, Australia
lin.padgham@rmit.edu.au

Sebastian Sardina
RMIT University
Melbourne, Australia
sebastian.sardina@rmit.edu.au

Sandip Sen
University of Tulsa
USA
sandip@utulsa.edu

ABSTRACT

Belief, Desire, and Intentions (BDI) agents are well suited for complex applications with (soft) real-time reasoning and control requirements. BDI agents are adaptive in the sense that they can quickly reason and react to asynchronous events and act accordingly. However, BDI agents lack learning capabilities to modify their behavior when failures occur frequently. We discuss the use of past experience to improve the BDI agent's behavior. More precisely, we use past experience to improve the context conditions of the plans contained in the plan library, initially set by a BDI programmer. First, we consider a deterministic and fully observable environment and we discuss how to modify the BDI agent to prevent re-occurrence of failures, which is not a trivial task. Then, we discuss how we can use decision trees to improve the agent's behavior in a non-deterministic environment.

Categories and Subject Descriptors

I.2.11 [Artificial Intelligence]: Distributed Artificial Intelligence—*Intelligent agents*; I.2.6 [Artificial Intelligence]: Learning

General Terms

Algorithms, Theory, Experimentation

Keywords

BDI, learning

1. INTRODUCTION

It is widely believed that learning is a key aspect of intelligence as it enables adaptation to complex and changing environments. Agents developed using the Belief-Desire-Intention (BDI) framework are capable of simple adaptations to their design-time behavior. First, such agents use hierarchical plans, where plan choices at each level are made in response to the current situation. Second, if a plan fails, often because the environment has changed since the plan was selected, agents backtrack and choose a different strategy for a particular level. However, BDI agents are unable to learn new behaviors

from their experiences, and hence cannot significantly alter their behavior from the one specified during their initial deployment. In this work, we analyze BDI-based agent designs and identify opportunities and mechanisms for introducing behavior learning into BDI agents, so as to allow them to better adapt to their current environment *on the basis of analysis of experiences*.

Research in machine learning can be broadly categorized into *knowledge-rich* and *knowledge-lean* techniques. Whereas some researchers have proposed and investigated learning mechanisms that incorporate and utilize significant amount of domain knowledge [8, 10, 18], the large majority of popular learning techniques assume very little domain knowledge and are largely data, rather than model, driven [1, 3, 17, 19, 24, 28]. Research in multiagent learning [2, 22, 32] has also followed this trend. This is particularly unfortunate as practical multiagent systems should be designed to leverage existing domain knowledge in order to facilitate scalability, flexibility, and robustness. For most such online, real-time multiagent systems, individual agents need to quickly and effectively respond to unforeseen events as well as to gradual changes in working and environmental conditions. The amount of experience and adaptation time available will be orders of magnitude less than what is assumed in offline, knowledge-lean learning algorithms. Mechanisms that leverage domain knowledge that aids and guides the learning and adaptation process will be key to the development of successful agent learning approaches.

In the context of BDI agents, we can foresee significant synergistic possibilities for combining *learning* and *reasoning* mechanisms. While on the one hand encoded domain knowledge of BDI agents can inform and direct embedded learning modules, the latter can incrementally adapt and update components of the reasoning module to tune agent's behaviors. Another possibility is for the learning module to use experience to refine coarse, approximate decision heuristics provided by the agent designer to produce improved behaviour over time.

BDI agents are very effective *practical reasoning* agents, able to quickly reason and react to asynchronous events [12, 27, 26]. Roughly speaking, a BDI agent senses events from the environment it is situated in, and through deliberation and means-end reasoning, selects appropriate courses of actions to be carried on. BDI agents are well suited for complex applications with (soft) real-time reasoning and control requirements [13, 15, 16, 20]. If the environment is changing over time, however, and design-time knowledge is no longer appropriately tailored to the situation, then performance can

degrade. What is more, it may be the case that while substantial knowledge is encoded, there are additional *nuances* which can be learnt that will result in better overall performance.

A critical issue for learning agents is how to adequately solve the *exploration* versus *exploitation* dilemma. Without appropriate prior knowledge, exploration is often blind, e.g., exploring agents choose actions from a uniform prior. Such uninformed exploration often compromises performance and can be arbitrarily costly. Available domain knowledge may help limit such losses by constraining action choices and focusing exploration on potentially useful regions of the action space. The information about useful areas for exploration could be encoded in the BDI plan library.

In the following section, we first provide an overview of the relevant aspects of typical BDI-style agents. We then discuss modifications to the BDI framework so as to include mechanisms for agents to improve their knowledge of which plans may be used in different situations. We do so by relying on a number of simplifying assumptions which make the scenario an “ideal” one. In Section 4, then, we outline ways in which these assumptions may be lifted.

2. BDI AGENTS

The BDI (Belief-Desire-Intention) model is based originally on the philosophical work of Dennet [9] and Bratman [5]. There is a large body of work in computer science that explores logics for such systems (e.g. [7, 25, 26]) as well as a substantial number of implemented systems (e.g. PRS [11], JAM [15], JACK [6], 3APL [14], Jason[4]). Basically BDI agents have a set of *beliefs* that represent the agent’s knowledge about the environment and about its own internal state; a set of *desires* or more commonly *goals* (non conflicting desires which the agent has decided to work towards achieving); and *intentions* regarding how the agent has decided to achieve its goals. In systems the intentions are generally a set of partially instantiated *plans* whose templates are part of the agent’s *plan library*.

An important aspect of BDI systems is that they do not plan from first principles. They have a library of hierarchical pre-defined plans or recipes which are selected based on suitability, and expanded as needed. They continually receive events from the environment, which can allow them to update their beliefs. Plans are then expanded based on current beliefs, making them extremely flexible and responsive to the environment. As a result they are well suited for complex applications with (soft) real-time reasoning and control requirements.

The architecture of a PRS agent is provided in Figure 1.

There are various versions of the basic BDI execution algorithm (e.g. [27, 34, 26]) a simple version of which is shown below:

While event queue and intention structure not nil do:

1. Select event (if any) *this may be an external event from the environment, or an event generated by the agent itself.*
2. Modify beliefs, goals, intentions. *New information may cause the agent to update its beliefs and modify its goals and intentions.*
3. Determine a set of “applicable plans” to respond to the new event. *Applicable plans are those tagged as relevant to the event type, whose context condition match the current beliefs, and which have not yet been tried for response to this event.*

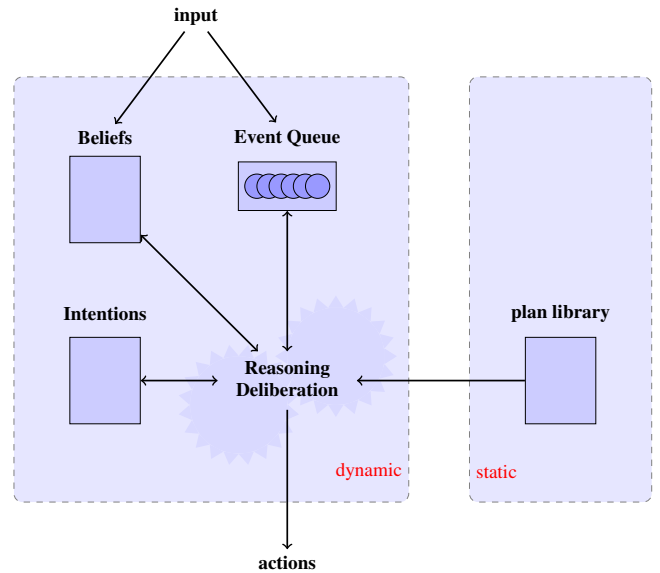


Figure 1: Architecture of the Procedural Reasoning System (PRS).

4. Select one applicable plan and add it to the intention structure. *This may be a new intention or it may expand the details of a current intention.*
5. Execute the next step of a selected intention. *This may execute an action or generate a new event.*

The plan library contains plans of the form $e : \psi \leftarrow P$ where P is the body of the plan, e is an event that triggers the plan, ψ is the context for which the plan can be applied (which corresponds to the preconditions of the plan). A plan body typically contains both actions, (which change the state of the environment), and subgoals which are events which result in expanding the details of the plan, by selecting a new plan for that event. By grouping together plans which respond to the same event type, the plan library can be seen as a set of *goal-plan tree* templates where goal (or event) nodes have children that are the alternative plans for achieving the goal, and plan nodes have children nodes that are the subgoals of the plan (or actions). These structures (see e.g. figure 2) can be seen as “AND”/“OR” trees: for a plan to succeed all the subgoals and actions of the plan must be successful (“AND”); for a subgoal to succeed one of the plans to achieve it must succeed (“OR”).

When a plan step (an action or subgoal) fails for some reason, this causes the plan to fail, and an alternative applicable plan for its parent goal is tried. If there is no alternative applicable plan, the parent goal fails, cascading the failure and search for alternative plans one level up the goal-plan tree. The search for alternative applicable plans on failure enables these systems to robustly recover from many problems, particularly problems where something has changed in the environment, motivating a different selection of plan.

The structured information contained in the goal plan tree can also provide guidance to the learning module. In particular, consider the context condition of plans, which are critical for guiding the execution of the agent program. A plan will not be used in the current

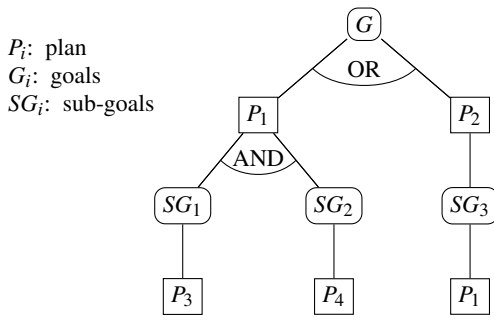


Figure 2: Goal/plan tree

state if its context condition is not satisfied. Incorrect or inadequate context conditions can lead to two types of problems. If the context condition of a plan is over-constrained and evaluates to false in some situations where the plan could succeed then this plan will simply never be tried in those situations, resulting in possible utility loss to the agent. On the other hand, if the context condition is under-specified, it may evaluate to true in some situations where the plan will be ineffective. Such “false triggers” will result in unnecessary failures, and although the agent may recover by choosing alternative plans, it may lose valuable time or waste resources unnecessarily thereby losing utility. Hence, it would be preferable to learn from experience to avoid using plans that are unlikely to succeed at particular environmental states.

The rest of this paper explores the issue of learning to improve the context conditions specified by the programmer.

3. PLAN SELECTION REFINEMENT

In order to set the scene for our discussion of approaches to learning improvements in the context condition, we explore in what ways plan selection could be refined based on experience, in an idealised and highly constrained setting. This setting allows us to describe and understand the ideal situation, noting that non-trivial reasoning is involved if we are to eventually exclude all unnecessary failures. The most straightforward way to refine plan selection is by gradually modifying context conditions of plans to make them more and more specific. However we also explore a more subtle refinement that could be done by a smart reasoner, regarding selection of plans sequences in specific situations. In section 4 we will relax the idealised constrained setting and show how we can apply learning techniques to refinement of context conditions.

Firstly we assume the the environment of the agent is deterministic. That is, if an action fails in a particular world state once, it will always fail when tried in this state. This allows us conceptually to have the possibility of making updates to context conditions only when we know that update is correct. Secondly we assume that the context condition as initially specified is definitely a necessary condition (i.e., a minimum condition for the plan to succeed). We will now discuss the situations in which we can with certainty update a context condition, or refine some more complex selection function, with the aim of eventually improving the selection to avoid any failure based on poor plan choice. We note that failure can also occur when the environment changes between the time of plan selection, and the time of executing some aspect of that plan. It is exactly this kind of failure which BDI systems are good at avoiding (by doing plan decomposition only when needed), and recovering from

(by trying an alternative plan if one fails). The latter mechanism also assists with managing the type of failure we are interested in learning to avoid.

In the following discussion we assume that the beliefs and the context conditions of plans are described using propositional logic. We also assume that all variables $v \in V$ that affect the functioning of the system are observable and known. In addition we assume that all changes to any variable v are explicit: that is we know if a particular action may (under some circumstances) change v . This is necessary for us to reason about changes caused by the agent, as opposed to those happening independently in the environment. Additionally we assume that the subset $\mathcal{R}_p \in V$ that is relevant for the success or failure of a particular plan p is defined. A likely candidate for \mathcal{R}_p can be identified as the set of variables that are accessed by some plan below the parent goal of p in the goal-plan tree. Although only a subset of \mathcal{R}_p may be used in the context condition of p , other variables in this set may play a role in the success or failure of p . Within plan bodies we allow only sequences of actions and subgoals, and also we impose the constraint that there is no interleaving of plans which are achieving concurrent goals. This removes the need to monitor for interactions between goals that could have an effect on success.

Let $r_p(t)$ be the tuple containing the values of \mathcal{R}_p at time t . If we have a plan, p with a single action a in its plan body, and this plan fails at time t , then definitely we should be able to update the context condition of p to exclude the partial state represented in $r_p(t)$. However, if p is a more complex plan there may be things (under the agent’s control) between t_s when p is selected, and t_f when p fails, that could have prevented that failure. In such a case modification of the context condition is not justified. The challenge is to know when it is impossible that failure could have been prevented (given the state $r_p(t_s)$ when p was selected), and therefore to know that we should modify the context condition of p to exclude $r_p(t_s)$.

Let us consider a plan p that is a sequence of actions, $a_1 \dots a_n$ and p has failed at step a_i . In this case, if none of the actions $a_1 \dots a_{i-1}$ affect variables in \mathcal{R}_p , we can be assured that the failure of a_i (and therefore p) was inevitable given $r_p(t_s)$ when p was selected. In this case modification of the context condition of p to exclude $r_p(t_s)$ is justified (as long as there have been no environmental changes to variables in \mathcal{R}_p , which can be checked by whether $r_p(t_s) = r_p(t_f)$). In fact, as we have assumed that actions are fully deterministic, it does not really matter if $a_1 \dots a_{i-1}$ affects variables in \mathcal{R}_p as they will always lead to the same state, which eventually caused failure. However, if we generalise actions to be sub-goals, then choices are potentially available, and we must consider this in our assessment of whether the failure was inevitable.

Consider the example situation shown by the plan-goal tree in figure 3. Assume that plans $p_{11}, p_{12}, p_{21}, p_{22}$ and p_{31} each consist of a sequence of one or more atomic actions (and are therefore deterministic). If p_{11} fails its context condition will be updated as discussed above. However, its failure will cause the failure of SG_1 , and therefore of p_0 . The question is when can we update the context condition of p_0 to exclude $r_{p_0}(t_s)$, the state when it was chosen and eventually led to failure. We cannot straightforwardly do this when p_0 fails, as possibly the choice of p_{12} to achieve SG_1 would have led to success. However if we have recorded that previously in a state equal to $r_{p_0}(t_s)$ we started executing p_0 , subsequently chose p_{12} to accomplish SG_1 , and this also failed, then we would be justified in updating the context condition of p_0 , to disallow $r_{p_0}(t_s)$.

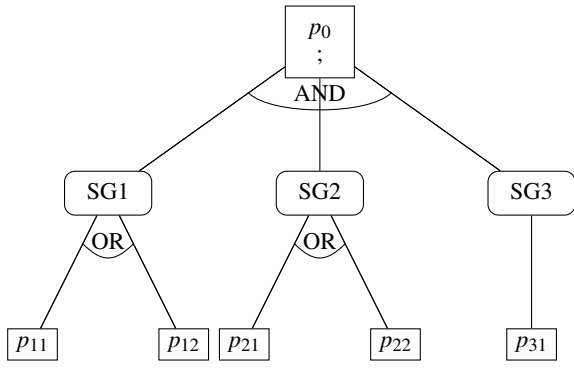


Figure 3: Example of a goal-plan tree.

This is because all possible execution paths from p_0 , chosen in state $r_{p_0}(t_s)$, have led to failure.

We now modify the situation slightly and assume the failure occurs initially at p_{31} , thus propagating to p_0 . In this case we can only know that the context condition of p_0 should be modified if all combinations of choices at SG1 and SG2 have been tried following selection of p_0 in state $r_{p_0}(t_s)$. However, there is some additional selection information that we could potentially use. If in the situation where we end up failing at p_{31} , we have used p_{11} to achieve SG1 and p_{21} to achieve SG2, then we know that this is a bad combination for the case where the state is $r_{p_0}(t_s)$ when we select p_0 . In future cases where we select p_0 in this state, we should avoid this combination of choices for achieving SG1 and SG2. However this is not information that can be represented as part of the context condition of a particular plan. Rather it is a complex metalevel selection function for choosing (or excluding) a path through the goal plan tree of a particular plan. It is not clear where or how such information should be represented in the BDI hierarchy, although it is clear that in an idealised world such information is potentially available.

One approach to determining whether a context condition should be updated following failure in a child node is to use planning. Once we have detected a failure we would like to propagate modifications as far up the Goal-Plan tree as is justified. Each time we have modified the context condition of a plan, we can check whether any plans one level up in the Goal-Plan tree should also be modified, using Hierarchical Task Network (HTN) plan decomposition, based on the BDI Goal-Plan tree as described in [29]. In our example, if after modifying the context condition of p_{31} we attempt plan decomposition on p_0 with initial state equal to $r_{p_0}(t_s)$, and fail to find a satisfactory decomposition, then we should modify the context condition of p_0 to exclude $r_{p_0}(t_s)$, as the plan decomposition has shown that there is no valid decomposition from this start state.

As we can see from the above discussion, in order to reliably update our knowledge about plan selection, we need to maintain information about all paths tried, from every relevant belief state combination, for all plans. Although some smart pruning may be possible based on knowledge of which paths can affect which variables, there is still a large amount of information that must be kept. Presumably compact representations such as those used by model checkers [] could also assist with practical feasibility.

However we have also made assumptions about a deterministic world and complete knowledge that are infeasible in practice. Therefore, based on the understandings gained from the discussion above we explore the use of learning to update context conditions, when, based on experience, it appears to be warranted.

4. LEARNING REFINEMENT OF CONTEXT CONDITIONS

The context in which we have discussed the refinement of plans' context conditions and the agent's plan selection function in the previous section can be seen as "ideal": (i) the environment is deterministic and fully observable; (ii) the (initial) context conditions are always necessary ones; and (iii) the success and failure of plans are always observable. Once we relax the condition of determinism and allow for non-deterministic actions (or lack of a fully observable environment, which is equivalent), we can no longer update any context condition based on a single failure as was the basic case for the previous section. Nevertheless, we should, over time, be able to learn that certain states do tend to lead to failure, and to refine our plan selection accordingly.

Decision Trees [21] are a natural choice of learning mechanism for this situation as they can deal with noise (created by the non-determinism in our case) and they support disjunctive hypotheses which we require. Also, a decision tree is readily convertible to rules, which are in fact the context condition.

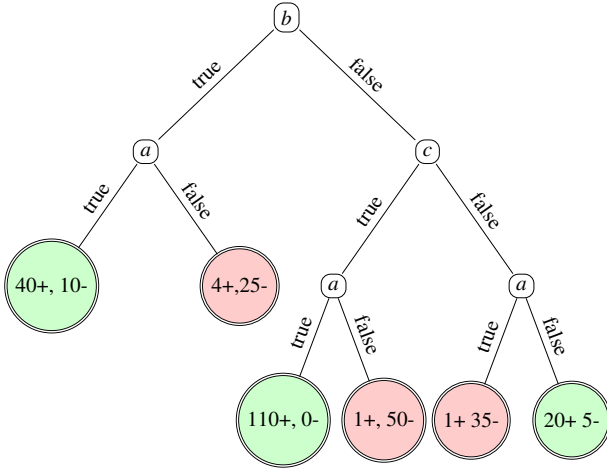
Others have also used decision trees as a tool in learning context condition details. In [23], inductive learning was used to build a decision tree to decide on which plan to use. The instances observed between the update were used to estimate the accuracy of the tree, and when the tree was successful enough, it was used as context condition. The problem faced by the learning agent is similar to ours when all initial context conditions are set to true. Their approach is tailored to a particular example, and it is not clear how to generalize it. Our goal is to provide learning capabilities to a generic BDI agent.

Typically, a decision tree is built off-line from a set of data. In our case though we wish to act as best we can, but accumulate experience to make improved decisions as we get more information. Some algorithms are designed to induce decision trees incrementally [30, 33], and we are planning to use these algorithms for our task.

We will associate a decision tree with each plan in our goal-plan tree. As we build up and record our experience, the decision tree will identify likely failure conditions, allowing us to modify our context condition in a similar way to our simplest example in the previous section. We will illustrate with an example the basic principles of our approach.

4.1 Example

Let assume that we have the goal-plan tree in Figure 5: the agent wants to satisfy the goal G and can choose between two plans: p_1 and p_2 . Let us assume that there are three relevant variables in \mathcal{R}_{p_1} and $\mathcal{R}_{p_2} = \{a, b, c\}$. First, let us assume that the context condition of both p_1 and p_2 is set to true, i.e., the programmer does not put any constraints on the applicability of either plan. As data is collected during the execution of the agent, a decision tree as in Figure 4 can be built for each plan. Once patterns seem to be clear (i.e., when not changes in the decision tree occur after seeing new instances)



Context condition converted from the decision tree :
 $(a \wedge b) \vee (a \wedge \neg b \wedge c) \vee (a \wedge \neg b \wedge \neg c)$.

Figure 4: Example of a decision tree for a plan p

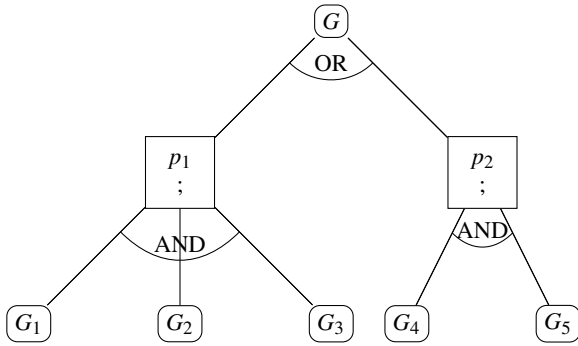


Figure 5: Example of a goal-plan tree.

conditions can be (provisionally) added to the context condition of the plan. For example, considering Figure 5 we may consider that we have enough information to add the condition $\neg(\neg b \wedge a \wedge \neg c)$ in a similar way to the information we added following a single failure in the previous section. Taking the information from the entire decision tree we can extract $(a \wedge b) \vee (a \wedge \neg b \wedge c) \vee (a \wedge \neg b \wedge \neg c)$ as the (probable) context condition for plan p_1 .

In order to build this decision tree for plan p , we will need to record the number of successes and failures in each possible partial state with respect to the variables in \mathcal{R}_p . If we have some initial constraints on the context conditions, given by the programmer, then not all points in the state space will record a result (as the constrained context condition will prevent the exploration of this space which is known to produce failure).

For example, let us now assume that the context conditions for the plans p_1 and p_2 are the following: $a \vee b$ for p_1 and $\neg a \wedge \neg b$ for p_2 . If we record our experiences in a table similar to Table 1, only part of Table 1 will be populated as the context condition prevents selection of some plans.

To build this table the agent must recall the state at which the plan was chosen, and then record the relevant success or failure in the

a	0	0	0	0	1	1	1	1
b	0	0	1	1	0	0	1	1
c	0	1	0	1	0	1	0	1
number of success								
number of failures								

Table 1: Table summarizing the failure and success of a plan p .

table. There will inevitably be some amount of noise in the data due to the non-determinism (or hidden variables) that we have introduced. This will occasionally cause a failure in a plan that generally works and is a good choice (for a given $r_p(t_s)$). However, another reason for plan failure (other than poor choice) is that there is a change to some variables in \mathcal{R}_p due to some environment factors outside the control of the agent. Such causes of failure should not lead us to conclude that P is a bad choice in $r_p(t_s)$. Consequently we do not record either the success or failure of any plan executions where a variable in \mathcal{R}_p is externally influenced after the selection of p .

4.2 Adaptive Context Conditions

So, let us first consider the case of *non-deterministic fully observable environments*, or what is equivalent, (deterministic) environments which are partially observable. Let us also consider that the context condition given by the BDI programmer are necessary. Under such settings, one can no longer rely on the fact that a plan would have the same success-failure outcome under exact belief states—the environment may behave differently or there are some hidden variables in the domain. Nonetheless, by monitoring the execution of such plan over time, we may be able to learn some “patterns” on its success and failure and refine its context accordingly.

As we assume that the context condition ψ set by the BDI programmer are necessary, the agent will use ψ and the decision tree to decide whether a plan is applicable. In addition, only a part of the table recording the number of successes or failures of a plan will ever be filled. In our example, only the cells in white can be filled for p_1 and only the cells in grey can be filled for p_2 . Consequently the table for each plan can be reduced to include only states not already excluded by the context condition. Note that in our example, the context conditions of the two plans do not overlap, i.e., the design of the BDI programmer is such that a single plan was applicable given a state of the environment. It is possible, however, that multiple plans are applicable, in which case there will be an overlap in the context conditions of the rules.

In a first phase, the agent uses its BDI decision mechanism, and in addition, it records the statistics in the table, and uses the instances to build the tree incrementally. The algorithm in [30] does not require to store all instances to grow the tree. Once ‘enough’ information is collected¹, the agent can start to use the decision tree in conjunction with the context condition encoded by the programmer. Each time the original context condition of a plan is satisfied, the decision tree is used to confirm whether the plan is indeed applicable or not. We now discuss a few scenarios on our example.

Let assume that each time c is true, the plan p_2 fails. After collect-

¹For now, we assume that the number of instances observed is above a threshold. This threshold depends on the size of the table that can be filled given the context condition

ing 'enough' evidence (we will discuss this issue later), the decision tree will consist of a single decision node with c and two leaf nodes to tell to use the plan when c is false, and not use it otherwise. The use of the decision tree will prevent the agent from choosing p_2 when c is true, and hence, this action should improve the behavior of the agent. As a side effect, if the plan p_2 was originally the only applicable plan when its context condition was satisfied, there is now no plan applicable for this state of the world. In our example, no plan is applicable when $(\neg a \wedge \neg b) \wedge c$.

Now, let us come back to the example of Figure 3, and let us assume that when the decision tree of plan p_{31} is used, there is a state of the environment where no plan is applicable for SG_3 , yielding the failure of plan p_0 . In Section 3, we described the condition to correctly update the context condition of p_0 when the context condition of p_{31} is changed, which requires careful examination. While learning the decision tree for p_0 , the precise reason for the failure of SG_3 is not important: whether SG_3 failed because p_{31} failed or because no plan was applicable for SG_3 , the result is the same: the goal SG_3 was not satisfied, and hence, p_0 failed.

As the agent acts, a decision tree is learnt for p_{31} , but also for p_0 . In particular, the latest decision tree will learn to avoid the contexts that lead to a context that cause SG_3 to fail. Actually, for the update of the decision tree of p_0 , it does not matter which sub-goal failed, what matters is the success or failure of the plan. However, what does matter is whether the failure of the sub-goal could have been avoided. For example, let us assume that, at time t_0 , SG_2 fails with the context condition set by the BDI programmer. Let us assume that at time t_c 'enough' data has been collected and the use of decision trees for p_{21} and p_{22} prevents failures of SG_2 . Between t_0 and t_c , the data collected for the plan p_0 will contain a set E of negative instances due to failure of SG_2 . The set E should now be considered as false negative, i.e., erroneous instances: the same instance would now be classified as a success given the correct decision trees for p_{21} and p_{22} .

A first solution to this problem is to start collecting data for the decision tree of p_0 only when the decision trees below are approximately correct. This is akin to the case where all paths had to be visited before making an update in Section 3. If we can use the analysis of Section 3 to propagate findings, it would speed up the learning process. For example, if the decision trees of the plans that satisfy SG_1 are accurate, as SG_1 is the first sub-goal, some knowledge could be propagate to the decision tree of p_0 . This may require a large total number of instances, especially if the depth of the tree is large. A different solution is to learn the decision trees concurrently, but wait longer before using them: as more data is collected after t_c , the impact of the initial false negative instances in E will decrease (as positive instances will be present) and the tree should be correct in the long run. This solution does not wait for bottom trees to be learnt before collecting data, so it may require a smaller total number of instances to perform well. However, there is a risk of the trees not to be correct when they are used too quickly.

4.3 Generalizing Context Conditions

We now consider the cases where it is not the case that the plan's context conditions given by the BDI programmer are always necessary conditions. In such cases, the agent may want, sometimes, to try a plan even if its context does not hold. For example, this may be a reasonable behavior when no plan is applicable for resolving a goal—it may be worth exploring the feasibility of some plan that looks currently not applicable. *A priori*, a BDI programmer should

avoid having the situation where no plan is applicable. However, if the agent uses the learning technique we proposed above to refine the context condition, the chances to have situations where no plan is applicable increases.

First, the table capturing the successes and failures of a plan cannot be restricted to the entries that match the context conditions: as the plan can be used even when the context condition is not satisfied, any cells can be updated. Now let us come back to the example of Figure 5, and let us assume that the context condition of p_2 is $(\neg a \vee \neg b) \wedge \neg c$ when we use the decision tree. When $(\neg a \vee \neg b) \wedge c$ occurs, no plan is *a priori* applicable.

The experience of the agent contains evidences that p_2 is not applicable. The alternative plan to satisfies the goal G would be to use the plan p_1 , but this plan is *a priori* not applicable, not because the agent has observed failures of p_1 , but because of the context condition set by the BDI programmer. Because of the particular environment, and because the programmer did not include the variable c in the expression of context condition of p_2 , it may be possible that the context of p_1 should be different and include the variable c .

Given the fact that no plan is applicable, we now have an incentive to try to use p_1 when $(\neg a \vee \neg b) \wedge c$ occurs. If p_1 also fails in that context, then, from experience, no plan is applicable. This exploration incurred a cost from the BDI agent: when $(\neg a \vee \neg b) \wedge c$ occurred, the plan p_1 failed as the agent did not follow the prior knowledge input by the BDI programmer. If, however, the plan p_1 succeeds, even though it was not supposed to, the agent will be able to satisfy G .

The exploration is triggered by the fact that, from its own experience and the context condition set by the BDI programmer, no plan is applicable to satisfy the goal G . In that case, the agent will consider all plans that satisfy goal G and the statistics about success or failure of the plan. If there exist some plans with no records of failure or success for that particular context, the agent will try one randomly. After having observed a 'sufficient' number of instances, a plan will be declared applicable or not. If the plan is declared applicable we must update the context condition by adding the conjunction, e.g. $\Psi \wedge (\neg a \vee \neg b) \wedge c$ for our example. Note that in the case where multiple plans can be explored, we could wait for all plans to have been 'sufficiently' tried and only update the context condition of the plans with the highest rate of success. For example, we can run a t-test to determine whether plans have a significantly different success rate and pick the ones with the highest one.

Hence, when the BDI programmer is not certain that the context conditions are necessary, we propose to use decision trees to refine or generalize the context conditions. The generalization can occur when the agent finds a state of the world where no plan is applicable (because of the context condition set by the BDI programmer or as a consequence of the refinement of a context condition). In that case, the agent will explore alternative plans and will determine, from experience, whether these plans are applicable or not.

4.4 Discussion

The question that remains is when and how to use the decision tree to refine the decision regarding whether a plan is applicable or not. There are two issues to consider. The first is the number of instances that are required to make an informed decision. The second is about the proportion of success: if a leaf node of the decision tree

contains 60% of success, should the plan be considered as applicable?

First, the agent must have observed a sufficient number of instances before starting to use the decision tree. Depending on the number of relevant variables and on the size of the tree below the plan, we can estimate how many examples are needed to make a decision with some confidence. For example, if a plan p has some sub-goals, the decision tree of p is probably approximately correct if the decision tree of the plans satisfying the sub-goals are correct.

If the agent observes failures and successes for a particular state of the world, when should the agent consider that a plan becomes

- applicable (when the context condition does not match) or
- no longer applicable (when the context condition matches but we observe a high rate of failures)?

Without knowledge of a utility function for satisfying the goal G , there is *a priori* no clear decision. If failure has a high cost, it may be preferable to do nothing. At this point, it is not possible to provide a generic argument. External feedback about the agent performance could later be used to tailor these decisions. For now, we envision to use a threshold value of the success rate (for example, an 80% success rate is required for a plan to be considered applicable).

In the example of Figure 3, in Section 3, we discussed how dependencies between different branches could affect success/failure at a particular node. For example it is possible that in state r_{p_0} , p_{31} fails because p_{11} was chosen to satisfy SG_1 . If p_{12} is used instead, p_{31} would succeed. We noted that while there is information that could potentially be reasoned about, to avoid future failure, this is part of a complex meta-level selection function related to the goal. These kind of dependencies between ways of achieving various sub-goals actually create problems for our decision tree learning beyond the lack of ability to learn the correct combination. They can cause the decision tree to learn an inappropriate applicability rule. For the decision tree of p_0 , let us assume that, when r_{p_0} occurs, p_{11} and p_{12} are chosen with equal probability. In that case, p_0 will obtain a 50% success rate. If p_{11} is chosen with a higher probability, p_0 could easily be considered not applicable in the context r_{p_0} . To avoid inappropriate assumptions in this kind of situation we need to at least be aware of dependencies between plans. We expect to be able to use a variation on the dependency summaries of [31] to address this problem.

5. DISCUSSION AND CONCLUSION

BDI agents are adaptive in the sense that they can quickly reason and react to asynchronous events and act accordingly. To implement a BDI agent, a programmer must provide a plan library and a condition telling when each plan is to be used. When the environment is complex, when the success and failures of plans depends on complex conditions, writing correct context condition for each plan may be a difficult task. As BDI agents lack capabilities to modify their behavior when failures occur frequently, we propose to analyse past experience to improve the BDI agent's behavior. More precisely, we propose to use past experience to improve the context conditions of the plans contained in the plan library, initially set by a BDI programmer.

We discussed how to modify the BDI agent to prevent re-occurrence of failures a deterministic and fully observable environment when the BDI programmer set necessary conditions for the success of each plan. We showed that even with these simplifying assumptions, correctly refining the context condition is not a trivial task. Then, we relax the assumption of a deterministic environment and we discussed how to use decision trees to improve the agent's behavior. Finally, we relaxed the assumption that the initial context conditions are necessary (in a specific environment, the applicability of a plan may be more general, the BDI programmer may have over constrained a the applicability of a plan). In that case, the agent may have to perform some guided exploration: when no plan is applicable, plans that are not applicable and that have not been tried under specific conditions can be executed.

The objective of the learning is for the agent to avoid re-occurrence of failures by using past experience to refine (and generalize) the context conditions of the plans. We are currently implementing an hybrid BDI-learning agent and a testbed scenario inspired from *RoboRescue*. We plan to first explore the case where the initial context conditions are necessary and decision trees are used to refine the context conditions. The issues we plan to consider include when can we start to collect data, when can we start to use the trees, and how many instances are needed to converge to a satisfying behaviour.

6. REFERENCES

- [1] D. W. Aha, D. Kibler, and M. K. Albert. Instance-based learning algorithms. *Machine Learning*, 6(1), 1991.
- [2] E. Alonso, M. d'Inverno, D. Kudenko, M. Luck, and J. Noble. Learning in multi-agent systems. *Knowledge Engineering Review*, 16(3), 2001.
- [3] L. Booker, D. Goldberg, and J. Holland. Classifier systems and genetic algorithms. *Artificial Intelligence*, 40:235–282, 1989.
- [4] R. H. Bordini, J. F. Hübner, and M. Wooldridge. *Programming Multi-agent Systems in AgentSpeak Using Jason*. Wiley, 2007. Series in Agent Technology.
- [5] M. E. Bratman. *Intentions, Plans, and Practical Reason*. Harvard University Press, Cambridge, MA, 1987.
- [6] P. Busetta, R. Rönquist, A. Hodgson, and A. Lucas. JACK Intelligent Agents - Components for Intelligent Agents in Java. Technical report, Agent Oriented Software Pty. Ltd, Melbourne, Australia, 1998. Available from <http://www.agent-software.com>.
- [7] P. R. Cohen and H. J. Levesque. Intention is choice with commitment. *Artificial Intelligence*, 42:213–261, 1990.
- [8] G. DeJong and R. Mooney. Explanation-based learning: An alternative view. *Machine Learning*, 1:145–176, 1986.
- [9] D. C. Dennett. *The Intentional Stance*. MIT Press, 1987.
- [10] T. Ellman. Explanation-based learning: A survey of programs and perspectives. *ACM Computing Surveys*, 21(2):163–221, November 1989.
- [11] M. Georgeff and F. Ingrand. Decision Making in an Embedded Reasoning System. pages 972–978, 1989.
- [12] M. P. Georgeff and A. L. Lansky. Reactive reasoning and planning. pages 677–682, 1987.
- [13] M. P. Georgeff and A. S. Rao. A profile of the Australian artificial intelligence institute. *IEEE Expert*, 11(6):89–92, 1996.
- [14] K. V. Hindriks, F. S. D. Boer, W. V. D. Hoek, and J.-J. C. Meyer. Agent programming in 3apl. *Autonomous Agents and*

- Multi-Agent Systems*, 2(4):357–401, 1999.
- [15] M. J. Huber. JAM: A BDI-theoretic mobile agent architecture. In *Proceedings of the Third International Conference on Autonomous Agents*, (Agents'99), pages 236–243, Seattle, WA, May 1999.
- [16] F. F. Ingrand, M. P. Georgeff, and A. S. Rao. An architecture for real-time reasoning and system control. *IEEE Expert*, 7(6), 1992.
- [17] L. Kaelbling, M. L. Littman, and A. W. Moore. Reinforcement learning: A survey. *Journal of AI Research*, 4:237–285, 1996.
- [18] J. Kolodner. *Case-based reasoning*. Morgan Kaufmann, San Mateo, CA, 1993.
- [19] P. J. Krause. Learning probabilistic networks. *The Knowledge Engineering Review*, 13(4):321–351, 1998.
- [20] P. Lokuge and D. Alahakoon. Improving the adaptability in automated vessel scheduling in container ports using intelligent software agents. *European Journal of Operational Research*, 177, March 2007.
- [21] T. M. Mitchell. *Machine Learning*. McGraw Hill, 1997.
- [22] L. Panait and S. Luke. Cooperative multi-agent learning: The state of the art. *Autonomous Agents and Multi-Agent Systems*, 11(3):387–434, 2005.
- [23] T. Phung, M. Winikoff, and L. Padgham. Learning within the bdi framework: An empirical analysis. In *Knowledge-Based Intelligent Information and Engineering Systems: 9th International Conference, KES2005, LNAI Volume 3683*, page 282, Melbourne, Australia, September 2005. SpringerVerlag.
- [24] R. J. Quinlan. Induction of decision trees. *Machine Learning*, 1:81–106, 1986.
- [25] A. Rao and M. Georgeff. Decision procedures for BDI logics. *Journal of Logic and Computation*, 8(3):292–342, June 1998.
- [26] A. S. Rao. AgentSpeak(L): BDI agents speak out in a logical computable language. In W. V. de Velde and J. Perrame, editors, *Agents Breaking Away: Proceedings of the Seventh European Workshop on Modelling Autonomous Agents in a Multi-Agent World (MAAMAW'96)*, pages 42–55. Springer Verlag, Jan. 1996. LNAI, Volume 1038.
- [27] A. S. Rao and M. P. Georgeff. An abstract architecture for rational agents. In C. Rich, W. Swartout, and B. Nebel, editors, *Proceedings of the Third International Conference on Principles of Knowledge Representation and Reasoning*, pages 439–449, San Mateo, CA, 1992. Morgan Kaufmann Publishers.
- [28] D. Rumelhart, G. Hinton, and R. Williams. Learning internal representations by error propagation. In D. Rumelhart and J. McClelland, editors, *Parallel Distributed Processing*, volume 1. MIT Press, Cambridge, MA, 1986.
- [29] S. Sardina, L. P. de Silva, and L. Padgham. Hierarchical planning in BDI agent programming languages: A formal approach. pages 1001–1008, 2006.
- [30] E. Swere, D. Mulvaney, and I. Sillitoe. A fast memory-efficient incremental decision tree algorithm in its application to mobile robot navigation. In *Proceedings of the 2006 IEEE/RSJ International Conference on Intelligent Robots and Systems*, pages 645–650, 2006.
- [31] J. Thangarajah, L. Padgham, and M. Winikoff. Detecting and avoiding interference between goals in intelligent agents. In *Proceedings of the 18th International Joint Conference on Artificial Intelligence (IJCAI 2003)*, pages 721–726, Aug. 2003.
- [32] K. Tuyls and A. Nowé. Evolutionary game theory and multi-agent reinforcement learning. *The Knowledge Engineering Review*, 20(1):63–90, March 2006.
- [33] P. E. Utgoff, N. C. Berkman, and J. A. Clouse. Decision tree induction based on efficient tree restructuring. *Mach. Learn.*, 29(1):5–44, 1997.
- [34] M. Wooldridge. *An Introduction to MultiAgent Systems*. John Wiley & Sons (Chichester, England), 2002. ISBN 0 47149691X, <http://www.csc.liv.ac.uk/~mjlw/pubs/imas/>.