Learning Context Conditions for BDI Plan Selection

Tracking Number: 9999

ABSTRACT

Belief, Desire, and Intentions (BDI) agent-oriented programming is a popular paradigm for developing complex applications with (soft) real-time reasoning and control requirements. BDI systems rely on *plan libraries* to achieve *goals*, and on *context sensitive* subgoal selection and expansion. While context conditions do provide the ability for intelligent plan selection, they are fixed at design time and do not allow agents to adapt to new environments. We extend the standard BDI programming framework to accommodate a more flexible and dynamic representation for context conditions and a (probabilistic) plan selection function that balances both exploration and exploitation of plans. We then develop and empirically investigate an aggressive and a conservative approach to learning context conditions. We demonstrate that a more conservative approach that takes account of failure only when it believes sufficiently informed decisions were taken is more robust. While each approach has a small advantage in some situations, an aggressive approach can incur high penalties.

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 under the Belief-Desire-Intention (BDI) approach [3] are capable of *simple* adaptations to their behaviors, implicitly encoded in their *plan library* (a collection of pre-defined *hierarchical plans* indexed by goals and representing the standard operations of the domain). This adaptation is due to the fact that *(i)* execution relies entirely on *context sensitive subgoal expansion*, and therefore, plan choices at each level of abstraction are made wrt to the *current* situation; and *(ii)* if a plan happens to fail, often because the environment has changed unexpectedly, agents "backtrack" and choose a different plan-strategy.

However, BDI-style agents are generally unable to go beyond such level of adaptation. Both the actual behaviours (the plans) and the situations for which they are appropriate are fixed at deployment time. It is possible to use meta-level reasoning regarding selections of plans for particular situations, which may provide substantial flexibility, but there is no principled learning. In this paper we explore the details of how effective plan selection can be learnt, based on experience of what works in which world states. This mechanism can then be applied to either refine existing context conditions for plan selection, or even to learn the context conditions from scratch as we have done in our experimental work.

In [1] the authors detail why it may be problematic for learning if the BDI hierarchical structure is not taken into account. They showed why it can be problematic to assume a mistake at a higher level in the hierarchy, when a poor outcome may have been related to a mistake in selection further down. However their empirical work suggested that perhaps the advantages of an aggressive approach, where all failures were considered meaningful, outweighed the advantages of careful consideration.

In this work, we have used a more sophisticated algorithm to determine when a decision should be considered sufficiently informed to record a mistake if it fails to produce a positive outcome. We have also analysed the kinds of structures which will benefit from careful consideration vs aggressive learning and demonstrate empirically the relative advantages of each. Finally, we have considered the situation where plans that are unlikely to succeed (based on current information) are excluded from consideration, and have shown that, in this scenario, the aggressive approach may never learn, whereas the careful consideration is much more robust.

The issue of combining learning and deliberative approaches for decision making of autonomous systems has not been widely addressed. Learning has been used to learn independent low level skills (for example learn how to pass a ball for robots in RoboCup soccer (e.g., [9]), which are then used by the deliberative decision making center of the agent. In our work, what is learnt from experience is actually used by the deliberative component of the BDI agents. Initially, instead of having a blind exploration strategy (as is the case when agents are learning with no initial knowledge), our agent has an optimistic strategy: it trusts the existing BDI hierarchy, in other words, our agent uses existing domain knowledge and keeps on improving it.

We believe this approach, which takes a BDI program and "tune it" using learning, can be very powerful. While substantial knowledge is encoded at design and implementation of the system, there are often additional *nuances* which can be learnt over time. Particularly in some environments, it may be very difficult to predict in advance, exactly how certain aspects of the world affect goal achievability. Our approach allows for encoding what we know, and learning the rest.

The rest of the paper is organized as follows. In the next section, we provide a quick overview of typical BDI-style

Cite as: Learning Context Conditions for BDI Plan Selection, All Authors, *Proc. of 9th Int. Conf. on Autonomous Agents and Multiagent Systems (AAMAS 2010)*, van der Hoek, Kaminka, Luck and Sen (eds.), May, 10–14, 2010, Toronto, Canada, pp. XXX-XXX.

agents. We then discuss modifications to the BDI framework to accommodate learning capabilities by extending context conditions and the agent plan-selection function. After that, we describe two approaches to learning the context condition of plans. We then report on the empirical results obtained and conclude that care must be taken to achieve robust and successful learning in a typical BDI hierarchical program. We conclude with a brief summary and discussion.

2. BDI PROGRAMMING

BDI agent-oriented programming is a popular, well-studied, and practical paradigm for building intelligent agents situated in complex and dynamic environments with (soft) real-time reasoning and control requirements [5]. BDI systems enable *abstract plans* written by programmers to be combined and used in real-time, in a way that is both flexible and robust. There are a plethora of agent programming languages and development platforms in the BDI tradition, such as PRS [5], Jack [4], 3APL [6], and Jason [2], among others.

In a BDI-style system, an agent consists, basically, of a belief base (akin to a database), a set of recorded pending events, a plan library, and an intention base. While the belief base encodes the agent's knowledge about the world, the pending events stand for the *goals* the agent wants to achieve/resolve. The *plan library*, in turn, contains *plan rules* of the general form $e : \psi \leftarrow P$ encoding the standard domain operation-strategy P (that is, a program) for achieving the event-goal e when so-called *context condition* ψ is believed true—program P is a reasonable strategy to resolve event e whenever ψ is believed to hold. The plan-body program P typically contains actions (act) to be performed in the environment, belief updates ($+b$ and $-b$ to add or delete a proposition, respectively), tests ($?\phi$) operations, and subgoals ($!e$) to post internal subgoal-events which ought to be in turn resolved by selecting suitable plans for that event. Lastly, the intention base accounts for the current, partially instantiated, plans that the agent has already committed to in order to handle or achieve some event-goal.

The basic reactive goal-oriented behavior of BDI systems involves the system responding to events, the inputs to the system, by committing to handle one pending event-goal, *selecting a plan* from the library, and placing its program body into the intention base. A plan may be selected if it is *relevant*, that is, it is a plan designed for the even in question, and it is *applicable*, that is, its context condition is believed true. In contrast with traditional planning, execution happens at each step. The assumption is that the use of plans' context-preconditions to make choices as late as possible, together with the built-in goal-failure mechanisms, ensures that a successful execution will eventually be obtained while the system is sufficiently responsive to changes in the environment.

For the purposes of this paper, we shall mostly focus on the plan library, as we investigate ways of learning how agents can make a better use of it over time. It is not hard to see that, by grouping together plans responding to the same event type, the plan library can be seen as a set of *goal-plan tree* templates: a goal (or event) node has children representing the alternative relevant plans for achieving it; and a plan node, in turn, has children nodes representing the subgoals (including primitive actions) of the plan. These structures, 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).

Consider, for instance, the goal-plan tree structure depicted in Figure 1. A link from a goal to a plan means that this plan is relevant (i.e., potentially suitable) for achieving the goal (e.g., P_1, P_1', P_2', P_3', and P_4' are the relevant plans for G); whereas a link from a plan to a goal means that the plan needs to achieve that goal as part of its (sequential) execution (e.g., P_1 needs to achieve first goal G_1 and then G_2). An edge with a label $\times n$ states that there are n edges of such type. Leaf plans are assumed to directly succeed or fail when executed in the environment, and are marked accordingly in the figure for some particular world. In some world, for our example, the agent may achieve goal G_2 by selecting and executing P_1^2 followed by selecting and executing 3 working plans, namely, P_1^a, P_1^b, and P_1^c to resolve goals G_a, G_b, and G_c, respectively. An analogous situation arises below plan P_1^1, but it is abstracted out in the figure for legibility. If the agent succeeds with goals G_1 and G_2, then it succeeds for plan P_1, achieving thus the top-level goal. There is no possible successful execution, though, if the agent decides to carry on any of the four plans P_i', $i \in \{1, \ldots, 4\}$, instead for achieving the top-level goal G.

The problem of *plan-selection* is at the core of the whole BDI approach: *which plan should the agent commit to in order to achieve a certain goal?* This problem amounts, at least partly, to what has been referred to as *means-end analysis* in the agent foundational literature [8, 3], that is, the decision of *how* goals are achieved. To tackle the plan-selection task, BDI systems leverage domain expertise by means of the context conditions of plans. However, crafting fully correct context conditions at design-time can, in some applications, be a demanding and error-prone task. In addition, fixed context conditions do not allow agents to adapt to new environments. In the rest of the paper, we shall provide an extended framework for BDI agent systems that allows agents to learn/adapt plans' context conditions, and discuss and evaluate two possible approaches for such learning task.

3. THE LEARNING FRAMEWORK

The learning task of interest is as follows: determine whether a plan is applicable in the current world state given past execution data, as well as conditions programmed by the developer. We use decision trees "attached" to each plan.

Decision trees [7] are a natural classification mechansim to choose for our purposes as they can deal with noise (potentially created by an environment with some non-determinism, or by other factors), and they are able to support disjunctive hypotheses. Also, they are readily convertible to rules, which are the standard representation for context conditions. Data is then collected regarding the world state at time of selection of the plan, and whether the plan is considered to have succeeded or failed. Success data is always recorded, whereas the recording of failure data may be subject to some analysis as described in the following section. For the real system we intend to use algorithms for incremental induction of the decision tree, such as those described in [10, 12], but for the purpose of our analysis we will use J48 from weka [13], a version of c4.5 [7], and recreate the decision tree after each new piece of data is collected. We also record the number of both successful and failed executions of the plan in the different leaf nodes. This information will be used to estimate the quality of the decision tree, as described below.

A decision tree will thus give us information for each relevant plan, in a particular world state, regarding how likely

it is to succeed/fail, based on the agent's experience so far. Given this information the agent must select which plan to try. There are various mechanisms for doing this in a standard BDI system, including plan precedence and meta-level reasoning, but all these mechanisms are pre-programmed and do not take into account the experience of the agent. However in the context of learning, we must consider the standard dilemma of exploitation vs exploration. We have taken a fairly simple approach to this which appears to work well. We select a plan using a probabilistic function, which selects a plan with a probability proportional to its relative success.

In the following work we make a number of assumptions to enable us to explore the basic issues we are concerned with, and to be able to more easily compare two extreme approaches. These assumptions are:

1. Actions in the environment may fail with some probability, even in world states where they "should" succeed. This captures the non-deterministic nature of most real world environments.

2. No changes happen in the environment, during execution, other than those made by the agent. (Although this may appear too limiting, the previous assumption of non-determinism mitigates against the oversimplification for many cases, by treating other changes as non-deterministic failures.)

3. The agent is only executing a single intention (i.e., pursuing a single goal). Without this assumption it would be necessary for the agent to reason about potentially negative interactions between concurrent goals. While such reasoning is possible [11], it complicates the situation for this work.

4. There is no automated failure handling, whereby the BDI system retries a goal with an alternative plan, if the selected plan fails. This will certainly be integrated into the system for real use, but again complicates the understanding of the basic mechanisms. In fact, its integration may help achieve faster learning.

4. CONTEXT LEARNING

With the classical BDI programming framework extended with decision trees and a probabilistic plan selection scheme, we are now ready to develop mechanisms for improving the selection of plans, based on experience. To that end, in this section, we develop two approaches for learning the context condition of plans.

What we are aiming to learn is which plans are best selected for achieving a particular goal, in the various world states that occur. Given that we have no measure of cost of particular plans,[1] a good plan for a particular world state is simply one which (usually) succeeds in that situation. Thus we need to store in our decision trees information about world states in which plans succeeded and failed. However, as we will show with an example, while it is always meaningful to record success, some failures may be better not recorded.

Consider the example in Figure 1. Suppose that in some execution, plan P_1 is chosen to address goal G, in some world state w_1; that subgoal G_1 is successfully executed; and that plan P_1^2 is chosen to try and achieve the second subgoal G_2,

[1]This could also be a useful addition, but is not part of standard BDI programming languages.

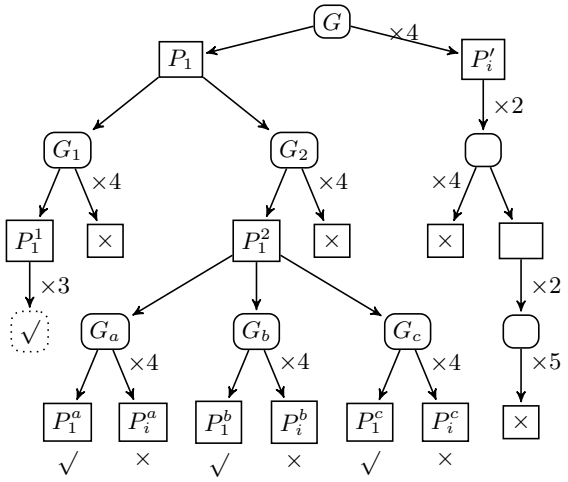


Figure 1: Goal-plan tree hierarchy $\mathcal{T}_3$. Rounded boxes stand for event goals; rectangles for plans.

requiring the achievement of G_a, G_b and G_c. Suppose further that G_a is successfully achieved using P_1^a, but execution *fails* when trying plan P_3^b for achieving subgoal G_b. As we have no failure recovery, this will immediately cause G_b to fail, and recursively will cause failure at each stage. Clearly, the failure should be recorded in the decision tree of the plan that caused the failure, namely, plan P_3^b. However, it is not so clear, as will be discussed next, whether the failure should be recorded in the decision trees for plans higher up in the hierarchy, that is, plan P_1^2 and even top-level plan P_1.

In order to discuss further which data should be recorded where, we define the notion of an *active execution trace* of the form $G_0[P_0 : w_0] \cdot G_1[P_1 : w_1] \cdot \ldots \cdot G_n[P_n : w_n]$, which represents the sequence of currently active goals, along with the plans which have been selected to achieve each of them, and the world state in which the selection was made—plan P_i has been selected in world state w_i in order to achieve the i-th active subgoal G_i. In our example, the trace at the moment of failure would be the following one:

$$\lambda = G[P_1 : w_1] \cdot G_2[P_1^2 : w_2] \cdot G_b[P_3^b : w_3].$$

When the final goal of λ fails, it is clear that the decision tree of the plan being used to achieve that goal ought to be updated, in the example, a failure should be recorded for the world state w_3 for the decision tree attached to plan P_3^b. Although it may be the case that the plan usually succeeds in the situation in which it was chosen, and failure is simply due to some non-determinism (or in the general case, actions of other agents, interactions, etc.), there is no way to determine this and the learning process will eventually recognise such cases as "noise." However, it is less clear whether the decision trees of plans associated with earlier goals in λ should be updated. It may be, in fact, that the failure of G_b could have been avoided, had the alternative plan P_1^b been chosen instead. If this is the case, then recording a failure against P_1^2 and P_1 is not justified. It may therefore be better to wait before recording failures against a plan until one is reasonably confident that subsequent choices were "well informed."

The judgment as to whether the decisions made were sufficiently "well informed," is however not a trivial one. A plan P is considered to be *stable* for a particular world state w

if the rate of success of P in w is changing below a certain threshold, ϵ. If the rate of success does not change much after some observations, we can start to build confidence about it. We also allow specification of a minimum number of execution experiences for P in w, in order to have the change of rate of success be sufficiently meaningful. A goal is considered to be stable for world state w if all its relevant plans are stable for w. We consider that the decision of recording a failure for a plan is "well informed" when the goal it is being chosen for is stable. We use a function StableGoal(G, w, k, ϵ) which returns true iff goal G is considered *stable* for world state w, under minimal number of executions and change of success rate thresholds $k \geq 0$ and $0 < \epsilon \leq 1$, respectively.

Algorithm 1 RecordFailedTrace(λ, k, ϵ)

KwData$\lambda = G_0[P_0 : w_0] \cdot \ldots \cdot G_n[P_n : w_n]$; $k \geq 0$; $\epsilon > 0$

RecordWorldDT$(P_n, w_n, failure)$
If[StableGoal$(G_n, w_n, k, \epsilon) \wedge |\lambda| > 1$] tcpdecision for G_n was informed $\lambda' \longleftarrow G_0[P_0 : w_0] \cdot \ldots \cdot G_{n-1}[P_{n-1} : w_{n-1}]$
RecordFailedTrace(λ', k, ϵ)

The algorithm starts by assimilating the failure in the last plan P_n in the trace, by recording the world w_n in which P_n was started as a "failure" case. If there is a previous node in the trace (i.e. for the plan P_{n-1} which required achievement of the failed goal G_n) and the choice of executing plan P_n to achieve goal G_n was indeed an informed one (that is, goal G_n was stable for w_n), the procedure is repeated for the previous node in the trace. If, on the other hand, the last goal G_n in the trace is not considered stable enough, the procedure terminates and no more data is assimilated. It follows then that, in order to update the decision tree of a certain plan, it has to be the case that the (failed) decisions taken during the plan execution must have been informed enough. The closer to 0 ϵ is, and the higher k is, the more conservative the agent will be in considering its decisions well informed. With $\epsilon =$ 1 and k = 0 we obtain a more standard learning approach where all information is accumulated, and the assumption is made that faulty information will eventually disappear as noise.

So, in the remaining of the paper, we shall compare two cases. The first we call aggressive concurrent learning (ACL) and is exactly the more traditional approach where all data is assimilated by the learner, that is, we take $\epsilon = 1$ and $k = 0$. The second we call bottom-up learning (BUL), and we use $\epsilon = 0.3$ and $k = 3$. We explore these two approaches on some programs with different structures and discuss the results.

5. EXPERIMENTAL RESULTS

In order to explore the difference between BUL and ACL we set up testbed programs composed of several goals and plans combined in a hierarchical manner with goal-plan trees of different depths and widths. We crafted hierarchical structures representing different meaningful cases of BDI programs, and made sure that there are always solutions, that is, successful executions of the top-level goal, if the right plan choices are made for a particular world state. We assume the agent is acting in a non-deterministic environment in which actions that are expected to succeed under certain conditions, may happen to fail with some (small) probability. In most of our experiments, we assume a .1 probability of uncontrolled failure for such actions.

The experiments consisted of posting the top-level goal repetitively under a random world state, running the corresponding Jack learning agent, and recording whether the execution terminated successfully or not. We calculated the average rate of success of the goal every 20 iterations, and investigate how this rate evolves as the agent refines the context condition of plans. We ran the tests using both a BUL-based agent and a ACL-based agent, ensuring the same sampling of random world states for each agent. With BUL, we used $k = 3$ and $\epsilon = 0.3$, meaning that the context condition of a plan is considered stable at a certain world, if we have recorded at least 3 past experiences and the rate of success has changed less than 0.3 in the last executions.

From our set of experiments, we have selected three hierarchical structures, that best illustrate the results that we obtained, and the reason behind these results.

(Tree $\mathcal{T}_1$) A structure in which ACL is expected to have important advantages aver BUL, since the former is capable of quickly assuming a top-level plan as not good, whereas BUL is expected to devote substantial more time to get to the same conclusion.

(Tree $\mathcal{T}_2$) A structure in which BUL is expected to have important advantages over ACL, since the latter may wrongly consider a top-level plan as a failing plan whereas there is a solution encoded under it.

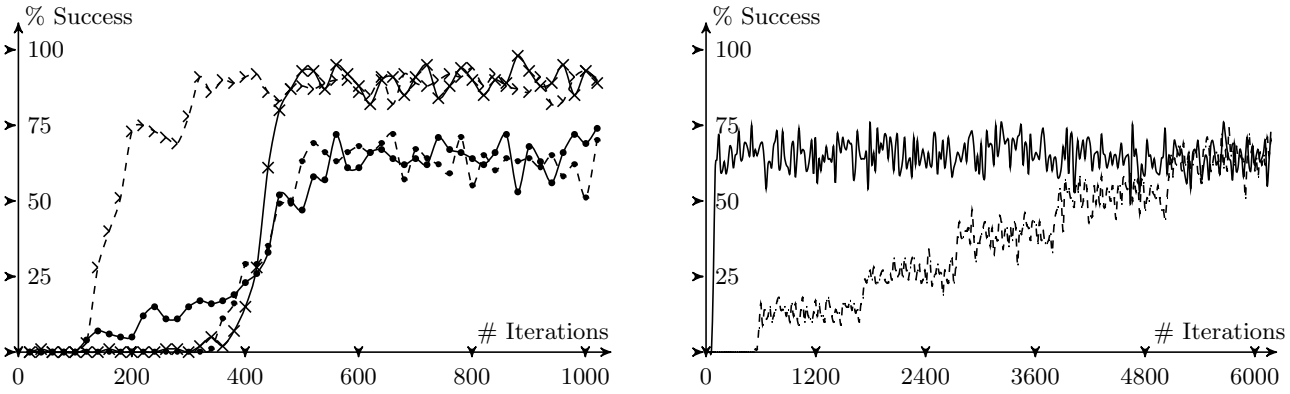
(Tree $\mathcal{T}_3$) A structure that is arguably a general case for BDI systems and that provides different advantages for both BUL and ACL in different parts of the tree.

In summary, we found that whereas the agent performance under the BUL and ACL approaches is comparable on the first and third cases, the BUL scheme provides substantial benefits in the second case. What is more important, if we consider agents that may choose not to consider a plan at all when its chances of success are believed very low, then the ACL approach collapses completely whereas the BUL is robust enough to maintain performance.

In Figure 3, we show the goal-plan tree structure $\mathcal{T}_1$. Initially, the agent has 20 options to resolve the top-level goal G. However, 19 of them lead to failure (plans P_i'). The benefit of using ACL comes from the fact that the agent will decrease the probability of selecting each of those 19 plans as soon as they fail for the first time. BUL however requires several failed experiences of each of those "bad" top-level options before decreasing their probability of selection—to update the decision tree of plan P_i', BUL would require each of their three subgoals to be "stable." As we expected the ACL scheme does perform better in that it achieves optimal performance faster (Figure 2(a), cross marked). The ACL approach yields optimal agent performance after around 300 iterations, whereas BUL achieves such performance after around 400 iterations[2].

The second structure $\mathcal{T}_2$ that we considered amounts to simplifying each plan P_i' to be just a single action that always fails and making the goal-tree hierarchy below plan P more complex, that is, deeper and with more goals.[3] Under such hierarchy, the agent needs to make four correct plan choices to result in a successful execution; there are

[2]Optimal performance amounts to a 90% success rate, as the environment fails with .1 probability and a successful run amounts to the execution of one action (a_1) only. If the environment is made fully deterministic, then agent performance eventually achieves complete success under both schemes.
[3]For lack of space, we do not show this structure.



(a) Structures $\mathcal{T}_1$ (crosses) and $\mathcal{T}_3$ (circles) (b) Structure $\mathcal{T}_2$

Figure 2: Agent performance under BUL and ACL schemes. Solid lines represent agent performance under the BUL approach; dashed lines represent agent performance under the ACL approach. Each point stands for the average of the last 20 executions.

also many chances for the agent to fail under P. Although one would expect BUL to yield better agent performance than ACL, the difference is enormous. Figure 2(b) shows that while the BUL approach, achieves optimal performance in around 100 iterations, the ACL scheme, requires around 5000 execution experiences. The reason is clear: since there are more chances to fail plan P initially, ACL marks this plan "bad," causing plans P_i' (which are all non-working) to be selected at least once in preference to trying P again. On the other hand, BUL would not consider P "bad" even when failing it, since it is aware that decisions made below were not informed enough. Consequently P will continue to be explored with equal likelihood to the P_i'. As a matter of fact, provided adequate parameters are used for checking stability, BUL would only record failed execution traces in plan P if the environment happened to fail unexpectedly: if the decisions were informed and the environment cooperated, then the execution is expected to succeed. Notice that optimal behavior amounts to less than a 75% rate of success, since the agent needs to ultimately perform four actions, each of them having a probability of success of 90% when performed in the real world.

Let us now consider the third hierarchical structure $\mathcal{T}_3$, depicted in Figure 1. In this case, the top-level goal G has five relevant plans, which are all "complex," that is, they all have several levels and multiple goals and plans. However, only one particular path in this hierarchy will lead to a successful execution for a particular world state. Among other things, this means that at the top-level the agent ought to select the right plan, all the other four plans are bound to fail eventually. We argue that this is the typical case in most BDI agent systems, in that for each goal, the agent may have several strategies, but each one is crafted to cover uniquely a particular subset of states. From the two learning approaches we are considering, structure $\mathcal{T}_3$ provides advantages for both of them, in different parts of the tree. The ACL scheme is expected to learn faster the inadequacy of the four non-working top-level programs, but the BUL would better explore, and find a solution, within the working top-level plan. This balance is corroborated by the fact that both agents have comparable performance, with BUL yielding improved behavior slightly quicker (see Figure 2(a), circle marked).

However, we are currently considering all plans as potentially applicable and worth trying - even when there is a very low chance of success. Given that executing a plan is often not cost-free in real systems, it is likely that the plan selection mechanism would in fact not execute plans with too low a probability of success. This would presumably hurt ACL. In order to demonstrate this we modified the probabilistic plan selection explained in Section 3 so that the Jack agent does not consider plans whose chance of success are below 0.2. In this experiment we also removed the non-determinism—actions always fail or succeed in each world state.[4] The differences, shown in Figure 2(b) are striking.

Using the structure $\mathcal{T}_3$ we found that whereas BUL maintains its performance (and in fact slightly improves, as failing leaf plans are discarded earlier than before thus reaching optimal performance around 100 iterations), the ACL approach is never able to learn and eventually is guaranteed to always fail the top-level goal.

The explanation for this wrong behavior under ACL is as follows. Initially, the agent tries all top-level plans for the top-level goal. Because of their complexity, it is extremely unlikely that the set of correct choices are made randomly. Thus their executions fail. This causes ACL to decrease the feasibility of all plans tried, including the top-level ones. As this will likely happen for several iterations, eventually all plans for a goal reach a probability of success lower than required, thus running out of options, failing the goal in question, and propagating the failure up in the hierarchy. Eventually, the top-level plans end up with low expectations of success and are ruled out. This will give no options to try and the goal will always fail.[5]

We conclude then that overall BUL clearly exhibits more

[4]Managing the non-determinism without continuing to, with some probability, try all plans, requires a more sophisticated mechanism than a simple probability check, to avoid randomly cutting off options that happen to have failed on one iteration. However in the interests of simplicity we deal with the simplified case.

[5]Such behavior does not arise in the original system because even if all plans have extremely low success chances, the agent would pick one anyway. As a result, the successful path would be eventually be found and plans' context conditions would start "recovering."

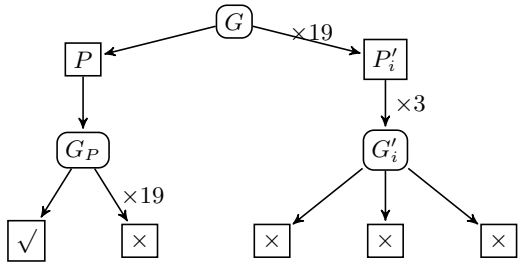


Figure 3: Goal-plan tree hierarchical structure $\mathcal{T}_1$. To succeed, an agent is thus required to make two correct choices, including selecting P at the top-level.

robust behaviour. In addition, in some structures ACL can pay significant costs, by considering some strategies as not viable too early. In those structures where BUL performs worse, the difference is relatively small.

6. DISCUSSION

In this paper, we proposed a technique to enhance the typical plan selection mechanism in BDI agent systems by allowing agents to learn and adapt the context condition of existing plans in the agent's plan library. As defining adequate context conditions for plans may be a difficult, and sometimes impossible, task, a framework that allows the *refinement* of (initial) context conditions would allow BDI agents to achieve better performance over time. The contributions of this paper are twofold. First, we develop an extended BDI framework in which learning mechanisms can be used. This is done using decision trees as (part of) context conditions, and developing a *probabilistic* plan selection function that caters for both exploration and exploitation of plans. Second, we propose and empirically evaluate two approaches for learning such decision trees based on agent executions. We found that, in general, the more conservative approach BUL, which learns from failed execution traces only when decisions were "substantially" informed, yields more robust agent performance than the simpler aggressive approach ACLwhich records every execution experience.

Our evaluation results were based on some important simplifying assumptions. In particular, we have not considered the effects of using failure recovery, under which alternative plans are tried upon the failure of a plan for a achieving a goal. Clearly, failure recovery would induce much less "pure" execution traces, as execution of partial plans would be represented in them. Also, we have focused on plan executability as the only objective. In some context, though, agents should also consider the *utility* of plans and expect to learn how to achieve better utility overall. One limitation of our learning framework is that the learning is *local* to plans, in the sense that it is not possible to learn interactions between sub-goals for a higher-level plan (e.g., learn that goal G_b may *only* succeed if goal G_a is achieved in a certain manner). Finally, we are planning to do further experimentation, in particular with some more realistic programs from existing applications.

7. INFORMED PLAN SELECTION

The first part of this paper addresses the task of determining when decisions along the active execution trace may be considered *informed enough* for outcomes to be included as learnable instances for each decision tree in that trace. For this we contrast two schemes, ACL and BUL - and show that the conservative BUL approach delivers more robust performance than the simpler ACL approach for the cases considered. For both approaches, the same probabilistic function E (or exploration strategy) was used to make the plan selection at each junction of the active execution trace.

In subsequent work we keep the ACL and BUL recording methods constant, and adjust the probabilistic plan selection function E to evaluate the impact on the learning outcome. Motivated by the polarity between the two approaches, we consider if an informed probabilistic selection function E' may be constructed to yield a "middle ground" approach that applies better in the general case. This would be valuable since if a ACL+E'-based approach yields comparable performance to any BUL+E^*-based approach, then the former is preferred as ACL is much simpler than BUL. Interestingly we find that such a formulation is possible and that an informed exploration strategy that leverages both the domain knowledge (inherent in the goal-plan tree hierarchy) as well as the ongoing learning (agent's experience so far) can combine the benefits of both ACL and BUL.

We start by quantifying the quality of (or our *confidence* in) the hypotheses of a decision tree for a unknown but learnable subspace S of worlds $[W_1 \dots W_n]$. At the beginning of a run we have no learnable instances and for a given goal-plan tree node T our probability of success $p_T(W)$ in world $W \in S$ is given by the default probability $P_d = 0.5$. After the first instance is recorded, the decision tree will generalise the result to the full space of S (i.e. $\forall W \in S$) leading to likely *misclassification*. Intuitively we know that as more and more $W \in S$ are witnessed and recorded, the decision tree's classification over S will improve. Our two approaches may be described in terms of confidence as follows: ACL always assumes full confidence in the decision tree but suffers from misclassification errors (that are eventually rectified through subsequent learning from misinformed choices); BUL uses the static ϵ and k values to determine the boundary for our confidence in the decision tree (with the trade-off that for the period of no confidence the best we can do is use P_d).

One way to improve this situation is to instead compute a smooth transition from zero-confidence in the decision tree classification to full-confidence, based on experience. We identify two orthogonal properties, *choices* and *subspace* that are integral to informed decision making by the learning nodes in our agent. Choices refer to the set of all statically computable execution paths *below* a goal node in the goal-plan tree hierarchy. Subspace refers to the set of all worlds that are learnable for the decision tree at that node. Defined in those terms, we label a decision tree *not informed* if no choices have been recorded in any world of the subspace, or *fully informed* when all choices have been covered in every world of the subspace. In fact, we can construct a measure of our confidence in the decision tree based on the degree of *coverage* of all choices in all subspace associated worlds.

$$E' : p'_T(W) = P_d + [c_T(S) * (p_T(W) - P_d)] \qquad (1)$$

Equation 1 shows how a confidence measure based on coverage may be used to modify the final plan selection probability. The idea is to bias the decision tree classification (probability of success) of world W in node T given by $p_T(W) \in [0 \dots 1]$ with the coverage $c_T(S) \in [0 \dots 1]$ of subspace S given $W \in S$. Initially the coverage is zero, so the revised probability $p'_T(W) = P_d$ the default proba-

bility of success. As coverage $\rightarrow 1$, the revised probability $p'_T(W) \rightarrow p_T(W)$ the decision tree classification. This gives a progressive transition to the decision tree classification as more experience is acquired (Figure 4).

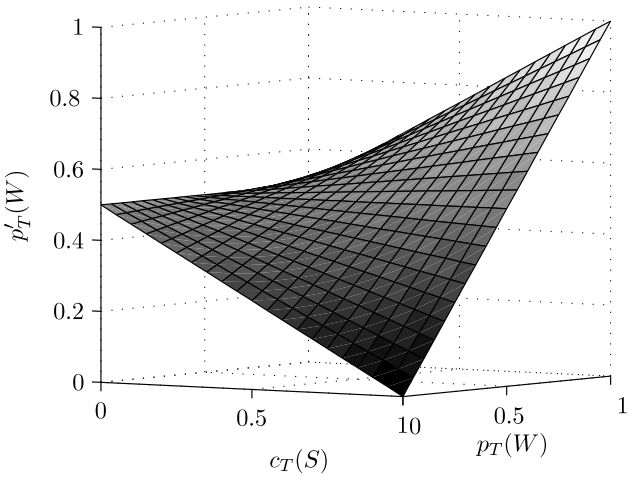


Figure 4: How coverage impacts decision tree classification

The coverage $c_T(S)$ itself is calculated and stored each time a result is recorded for node T. The calculation is performed in turn for each node in the active execution trace starting at the leaf node where the failure occurred, and the coverage is updated progressively up the tree hierarchy. Full coverage at a node T has a computation cost of $C(T) * |S|$ where $C(T)$ is the total number of choices below T and $|S|$ is the number of worlds in the subspace. Practically however, it costs significantly less since choices below T are effectively AND/OR trees, and each time an AND node fails the subsequent nodes are not tried and are assumed covered for that world. The full memory cost for a subspace with n boolean attributes is $2^a * C(T)$ however we only require 2^a for the implementation since we do not keep track of each individual path below T but only how many distinct paths below T have been tried in a given world. The only unknown in the coverage calculation is S since we do not know upfront the subspace to be learnt. A fairly useful estimate of $c_T(S)$ for practical use can however be constructed by averaging the individual coverages $c_T(W_i)$ of all previously seen worlds at node T.

8. REFERENCES

[1] S. Airiau, L. Padgham, S. Sardina, and S. Sen. Incorporating learning in BDI agents. In *Proc. of ALAMAS+ALAg*, 2008.

[2] R. Bordini, J. Hübner, and M. Wooldridge. *Programming Multi-agent Systems in AgentSpeak Using Jason*. Wiley Series in Agent Technology. Wiley, 2007.

[3] M. Bratman, D. Israel, and M. Pollack. Plans and resource-bounded practical reasoning. *Computational Intelligence*, 4(4):349–355, 1988.

[4] P. Busetta, R. Rönnquist, A. Hodgson, and A. Lucas. JACK intelligent agents - Components for intelligent agents in Java. *AgentLink News*, 2:2–5, 1999.

[5] M. P. Georgeff and F. F. Ingrand. Decision making in an embedded reasoning system. In *Proceedings of IJCAI*, pages 972–978, Detroit, USA, 1989.

[6] K. Hindriks, F. D. Boer, W. V. D. Hoek, and J. Meyer. Agent programming in 3APL. *Autonomous Agents and Multi-Agent Systems*, 2(4):357–401, 1999.

[7] T. Mitchell. *Machine Learning*. McGraw Hill,1997.

[8] M. E. Pollack. The uses of plans. *Artificial Intelligence Journal*, 57(1):43–68, 1992.

[9] M. Riedmiller, A. Merke, D. Meier, A. Hoffman, A. Sinner, O. Thate, and R. Ehrmann. Karlsruhe brainstormers - a reinforcement learning approach to robotic soccer. In *RoboCup 2000: Robot Soccer World Cup IV*, 2001.

[10] E. Swere, D. Mulvaney, and I. Sillitoe. A fast memory-efficient incremental decision tree algorithm in its application to mobile robot navigation. In *Proc. of IROS*, 2006.

[11] J. Thangarajah, M. Winikoff, L. Padgham, and K. Fischer. Avoiding resource conflicts in intelligent agents. In *Proc. of ECAI*, pages 18–22, 2002.

[12] P. E. Utgoff, N. C. Berkman, and J. A. Clouse. Decision tree induction based on efficient tree restructuring. *Mach. Learning*, 29(1):5–44, 1997.

[13] I. Witten and E. Frank. *Data Mining: Practical Machine Learning Tools and Techniques with Java Implementations*. Morgan Kaufmann, 1999.