Extending BDI Plan Selection to Incorporate Learning from Experience

Dhirendra Singh & Sebastian Sardina & Lin Padgham

RMIT University, Melbourne, Australia

Abstract

An important drawback to the popular Belief, Desire, and Intentions (BDI) paradigm is that such systems include no element of learning from experience. We describe a novel BDI execution framework that models context conditions as *decision trees*, rather than boolean formulae, allowing agents to *learn* the probability of success for plans based on experience. By using a probabilistic plan selection function, the agents can balance exploration and exploitation of their plans. We extend earlier work to include both parameterised goals and recursion and modify our previous approach to decision tree confidence to include large and even non-finite domains that arise from such consideration. Our evaluation on a pre-existing program that relies heavily on recursion and parametrised goals, confirms previous results that naive learning fails in some circumstances, and demonstrates that the improved approach learns relatively well.

Keywords:
BDI Intelligent Agents, Machine Learning, Hybrid Systems

1. Introduction

Agents are an important technology that have the potential to take over contemporary methods for analysing, designing, and implementing complex software systems suitable for domains such as telecommunications, industrial

Email address:
`{dhirendra.singh,sebastian.sardina,lin.padgham}@rmit.edu.au` (Dhirendra Singh & Sebastian Sardina & Lin Padgham)

control, business process management, transportation, logistics, and aeronautics [1, 2, 3, 4]. The BDI model of agency [5, 6] is a popular and well-studied approach with substantial theoretical and practical work. It has its roots in philosophy with Bratman's [7] theory of practical reasoning and Dennett's theory of intentional systems [8]. A recent industry study [9] analysing several applications claimed that the use of BDI (Belief-Desire-Intention) agent technology in complex business settings can improve overall project productivity by upto 500%. Also the agent approach allowed the business to change and extend solutions quickly helping to bridge the semantic gap between the business side and IT development. BDI systems have built into them an ability to balance pro-actively pursuing a goal, with reactively responding to the environment. They also have a well developed failure recovery mechanism. This makes them very suitable for robotics applications operating in a physical world which is often more error prone than a software domain.

BDI systems, despite their strengths, do not however incorporate any ability to learn from experience. Our work makes a start at addressing this issue, focussing specifically on learning which plan to select next, to resolve a particular goal in a particular world state.

There are many agent programming languages and development platforms in the BDI tradition, including JACK [10], JADEX [11], and Jason [12] among others. All of them follow a similar basic architecture, whereby *abstract plans* written by programmers are combined and used *reactively* in real-time, in a way that is both flexible and robust. Concretely, a BDI agent is built around a *plan library*, a collection of pre-defined *hierarchical plans* indexed by goals and representing the standard operational procedures of the domain (e.g., landing a plane). A *context condition* attached to each plan states the conditions under which the plan is a sensible strategy to address the corresponding goal in a given situation (e.g., it is not raining). The execution of a BDI system then relies on *context sensitive subgoal expansion*, allowing agents to "act as they go" by making *plan choices* at each level of abstraction with respect to the current situation. Although this is quite flexible and effective, an important drawback is the lack of ability to learn from ongoing experience. An ability to *learn* plan selection in particular situations, adds a whole new layer of robustness and flexibility. Firstly there may be situations where it is difficult to determine in advance the exact situation under which a particular approach is likely to succeed. This is especially the case when it involves complex combinations of values of environmental variables. Secondly, an environment may change over time, or be slightly different in different deployment locations. The ability for the agent system to observe and learn from its performance is obviously a very desirable property.

In our work we achieve this by replacing (or augmenting) the usual boolean formula for representation of context conditions, by a decision tree [13] which is learnt based on experience. Our plan selection is then based on a probabilistic approach, usually choosing the plan which has the highest likelihood of success, based on experience. This probabilistic approach is also more suitable than the standard boolean approach for complex and often partially observable worlds, where various plans may be worth trying, but have different chances of success.

There are however various nuances that must be addressed for such *online* learning. There is the usual balance between exploration and exploitation evident in all learning. Moreover, learning is impacted by the structure imposed by the hierarchical representation of BDI programs. We have addressed these issues in previous papers [14, 15], looking at various approaches to the problem. In this paper we add the ability to deal with paramaterised goals and with recursive calls, both of which are essential for real applications. Unfortunately, once we add this expressivity our previous preferred approach does not scale. Consequently we develop a simplified approximation to achieve the same basic intuition which we have previously shown to be correct in principle. We then empirically evaluate our approach by taking an existing BDI program from the JACK tutorial, removing the context conditions, and learning the appropriate use of the plans provided using our framework.

Our approach can easily be combined with the standard plan selection mechanism, by allowing the agent programmer to provide initial context conditions that could later be automatically "refined" by the agent system. By doing so, one can effectively take a BDI program and "tune it" using our learning framework. For simplicity, though, context conditions are learnt from scratch in our experimental work.

In the next section we introduce both BDI programming and our learning framework, as well as an overview of our previous approaches. We then describe in detail the learning framework that incorporates the additional aspects of parameterised goals and recursive calls, with our revised approach to address previously identified issues. We show empirical evaluation on an example program developed by Agent Oriented Software for their JACK tutorial, by removing the context conditions and applying our learning approach. We finish with a discussion of outstanding issues and related work.

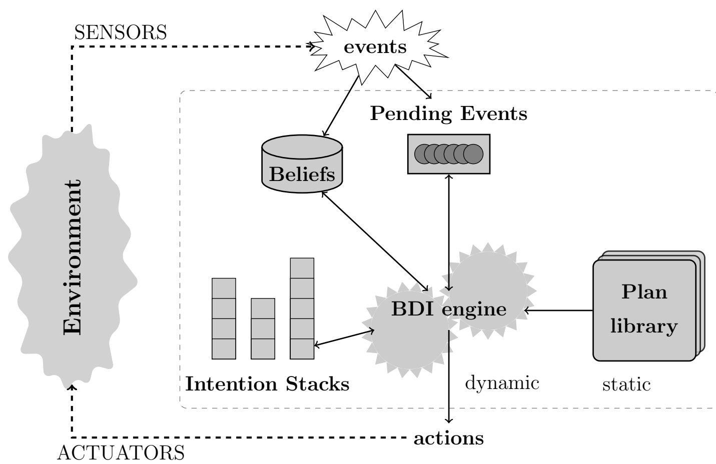


Figure 1: A typical BDI-style architecture

2. Preliminaries

2.1. BDI Agent Systems

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 [16, 9]. Generally speaking, BDI agent-oriented programming languages are built around an explicit representation of propositional attitudes (e.g., beliefs, desires, intentions, etc.). A BDI architecture addresses how these components are represented, updated, and processed to determine the agent's actions.

A BDI agent consists, basically, of a belief base (akin to a database) which stores the agent's knowledge about the world, a set of pending events, which includes both external percepts or messages and internal goals, a plan library and an intention structure. Figure 1 depicts a typical BDI architecture. The plan library contains rules of the form $e : \psi \leftarrow \delta$ indicating that δ is a suitable procedure for achieving event-goal e when context condition ψ is true. Among other operations, the plan body procedure δ will typically include the execution of actions (act) in the environment and subgoal events ($!e$) that are in turn resolved by selecting suitable plans for that subgoal event. The intention structure contains the current, partially instantiated, plans that the agent has committed to in order to achieve some event-goals.

The basic *reactive goal-oriented behavior* of BDI systems involves the

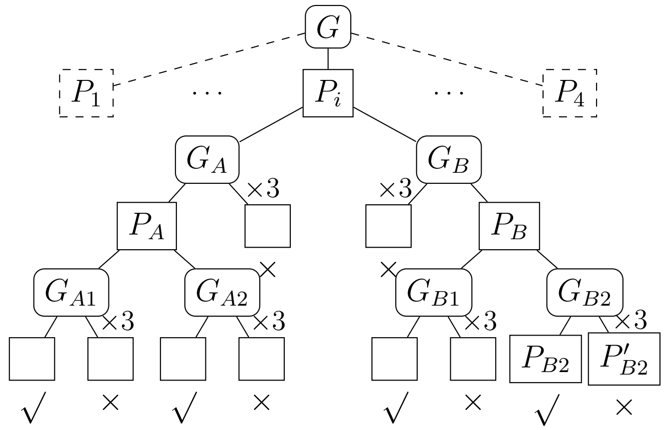


Figure 2: An Example BDI Goal-Plan Hierarchy.

system responding to events by selecting an appropriate plan from the library, and placing its program body into the intention base. A plan is appropriate if it is designed for the event in question (relevant) and its context condition is believed true (applicable). In contrast with traditional planning, execution happens at each step. The use of plans' context-preconditions to make choices as late as possible, together with the built-in goal-failure mechanisms, ensures that the system is responsive to changes in the environment. In this paper we focus on the plan library to investigate ways of learning appropriate or better plan selection based on experience.

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 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 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 hierarachical structure shown in Figure 2. A link from a goal to a plan means that this plan is relevant (i.e., potentially suitable) for achieving the goal (e.g., $P_1 \ldots P4$ are the relevant plans for event goal 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., plan

P_A needs to achieve goal G_{A1} first and then G_{A2}). For compactness, an edge with a label $\times n$ states that there are n edges of such type. Leaf plans directly interact with the environment and so, in a given world state, they can either succeed or fail when executed; this is marked accordingly in the figure *for some particular world* (of course, in other states, such plans may behave differently). In some world, given successful completion of G_A first, the agent may achieve goal G_B by selecting and executing P_B, followed by selecting and executing two working leaf plans to resolve goals G_{B1} and G_{B2}. If the agent succeeds with goals G_{B1} and G_{B2}, then it succeeds for plan P_B, achieving thus goal G_B and the top-level goal G itself. There is no possible successful execution though, if the agent decides to carry on any of the three plans labelled P'_{B2} for achieving the low-level goal G_{B2}.

As can be seen *plan-selection* is critically important. Standard BDI systems leverage domain expertise by means of the context conditions of plans. In this work, we are interested in exploring how a situated agent may *learn* plan selection based on experience, in order to improve goal achievement.

2.2. Learning for BDI Plan Selection

In order to facilitate learning regarding which plan should be executed for a given goal in a particular world state, we first replace the boolean formulae that are the standard representation for context conditions in BDI programming languages, with decision trees [13] that provide a judgement as to whether the plan is likely to succeed or fail for the given situation.

To select plans based on information in the decision trees, we use a probabilistic method whereby we choose a plan with probability proportional to its believed chance of success in the given situation. This approach provides a balance between exploitation (we choose plans with relatively higher success expectations more often), and exploration (we sometimes choose plans with lower success expectation to get better confidence in their believed applicability by trying them in more situations).

The resolution of goals in BDI execution results in the invocation of plans that in turn may post sub-goals that are further handled by sub-plans in a hierarchical manner. In a programming context, this is equivalent to making a function call that in turn calls sub-functions. However a sub-goal may have a number of possible plans for achieving it, some of which will work better in particular situations than others. In our learning context, where we do not yet know which plans work well in which situations, a plan may fail not because the plan was a bad choice in the given situation, but instead because the run-time choice of sub-plans was incorrect for the situation.

Our first approach [14] to address this problem was that of careful consideration whereby failures are recorded for learning purposes only when we are sufficiently sure that the failure was not due to poor sub-plan choices. We have shown that this conservative approach is more robust, though often slower, than a more agressive approach which records all experiences, but can in some particular cases completely fail to learn.

Our second approach reported in [15] was to adjust the plan selection probability based on some measure of our confidence in the decision tree classification. We consider the reliability of a decision tree for a given plan in a given world state to be proportional to the number of sub-plan choices (or paths below the plan in the goal-plan hierarchy) that have been *covered* in that world state. The greater the coverage, the more we have explored and the greater the confidence in the resulting decision tree. By biasing the plan selection probability with a coverage-based confidence measure we achieved the same robustness as that of conservative recording of failure cases. The coverage approach, however, is more flexible as the extent to which this is used can be readily adjusted by parameters in the selection formula.

A limitation with the previous approaches is that events were assumed to be propositional atoms i.e. event types were not considered. By *event type* we mean an event that may contain "data" as part of its definition. For instance, event *travelTo(?dest)* may represent the goal to travel to location *?dest*. Presumably a goal to move to location A would require different choices than a goal to move to B, and the learning must account for this. Another limitation is the assumption that the agent's plan library is non-recursive, so that the goal-plan tree structure induced is always *finite*. Clearly both limitations would preclude the applicability of the approach in many practical domains where hierarchies are usually expressed using parameterised goal events and plans, and often make use of (direct or indirect) recursive procedures.

Furthermore, the coverage approach of [15] does not scale when recursive structures are considered. Conceptually we can unfold the recursive structure to a specified depth. However, the number of paths is exponential in the recursion number and further compounded by event-goal types and the number of possible world states. An additional limitation is that coverage does not consider domain complexity. For instance, a leaf plan that has no subgoals will achieve full coverage when it is tried once, after which selection will be fully biased towards the plans decision tree classification. However, the decision tree that at this point has only witnessed one world will generalise the outcome to all as-yet-unseen worlds leading to misclassification.

In the following section we present the details of our learning approach, incorporating both parametrised goals, and recursion, as well as a new simpler confidence measure that is based on the general idea of coverage but does not suffer from its limitations.

3. The BDI Learning Framework

Our learning task may be summarised as follows: *Given past execution data and the current world state, determine which plan to execute next in order to best address the event-goal in question.* In the BDI sense, our task is to learn the context condition of each plan in the goal-plan hierarchy. In this section we describe our BDI Learning Framework that enables such learning. In particular we describe the use of decision trees for learning context conditions and the confidence-based probabilistic plan selection that incorporates this learning, while focusing on event-goal types and recursion.

3.1. Integrating Decision Trees into Context Conditions for Plans

In a BDI system, a plans context condition is a logical formula that is constructed at design time and evaluated against an event-goal at run time to determine if the plan is applicable in the given world state[1]. As reported in [14], in order to allow the context condition to be learnt over time, we annotate each plans context formula with a *decision tree*[2]. The idea is that the agent starts with some *necessary but possibly insufficient* conditions for each plan (provided by the designer), and over time and in the course of trying various plans in various world states will be able to *refine* each plans context condition using the learnt decision tree classification of the world states encountered.

The choice of decision trees as the learning module is motivated by several factors. Firstly, decision trees support hypotheses that are a disjunction of conjunctive terms, and since context formulas are generally expressed in this

[1]Context formulas may reference internal beliefs as well as environment states, and for this study we treat both as included in the world state.

[2]It is perfectly feasible to combine the existing logical formula with the decision tree classification, but to aid our understanding of the decision tree learning in this study we always use an empty initial formula.

form then decision trees are readily applicable. Secondly, decision trees can be converted to *if-then* rules that are human readable and can therefore be verified by a domain expert. Finally, decision trees are robust against training data that may contain errors. This is specially relevant in stochastic domains where applicable plans may nevertheless fail due to unforeseen circumstances.

The input for the decision tree learning is a training set of data points of the form $[w, o]$, where w is the world state in which the plan was executed and o was the boolean outcome (success or failure). Initially the training set is empty and grows over time as the agent tries the plan in various world states and samples the result. The world state w itself is a set of discrete attributes that together represent the state of affairs. The idea is that over time the decision tree will learn a classification based purely on the subset of attributes in w that are relevant to the context condition of the plan.

The attributes in w determine the quality of the final classification, and their number and possible values has a bearing on the size of the training set required to correctly learn the context condition. The choice of attributes to include in the world state w is a design decision and dependent on domain knowledge. Importantly, the attributes in w should be a superset of the necessary and sufficient attributes relevant to the context condition. For instance, for a plan to pick up an object using a robotic arm, *objectSurface* is a relevant attribute, *gripperWet* possibly is, but *dayOfWeek* likely is not. For our purposes we assume that the designer provides a set of all attributes that are considered possibly relevant to the context condition of the plan. In the worst case, this set is the full set of attributes of the world.

The decision tree inductive bias is a preference for smaller trees. In other words, the induction of decision trees will trade-off some accuracy in classification for compactness of representation. In fact such inductive bias is necessary if the decision tree is to generalise to as yet unseen world states. Once a decision tree is induced from the training set, it may be used to classify any new world state w. In the strict sense the classification is an outcome o (failure or success). However, several decision tree implementations including J48 in weka[3] annotate a likelihood of class membership (that is indicative of the inductive bias) to the returned classification. For the given world state w then, we treat the returned likelihood of membership to the *success* class

[3]In our study we use algorithm J48, a version of c4.5 [13], from the weka learning package [17].

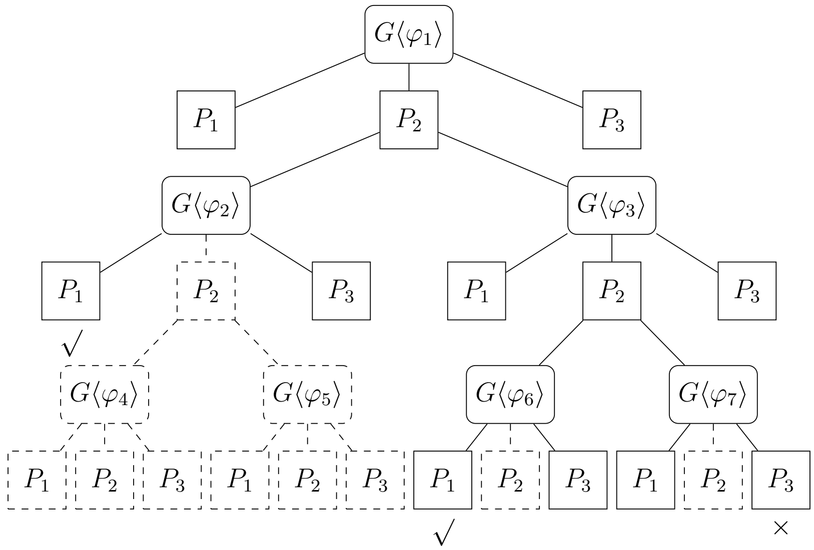


Figure 3: Goal-plan hierarchy containing a goal type $G\langle\rangle$ handled by three plans P_1, P_2 and P_3. Here plan P_2 posts two instances of $G\langle\rangle$ resulting in recursion. Two levels of recursive unfolding are shown. Dashed P_2 nodes indicate unexplored recursive sub-trees.

as the expected likelihood of success of the plan.

3.2. Support for Recursive Event-Goals

Recursion in our context refers to the case where the resolution of an event-goal instance $G[\varphi_1]$ involves first the resolution of another goal-event instance $G[\varphi_2]$ of the same type. The result is a growing stack of pending $G[\varphi_i]$ event-goals that eventually terminate in $G[\varphi_n]$ whose parameters satisfy the termination conditions where a non-recursive plan choice is made.

In order to understand the impact of recursion on context learning, we use the notion of an *execution trace* of the form $G_0\langle\varphi_0\rangle[P_0 : w_0] \cdot G_1\langle\varphi_1\rangle[P_1 : w_1] \cdot \ldots \cdot G_n\langle\varphi_1\rangle[P_n : w_n]$, that represents a sequence of event-goals along with the plans selected to handle them and the world state in which the selections were made. So $G_i\langle\varphi_i\rangle[P_i : w_i]$ captures the case where plan P_i was selected in world state w_i in order to achieve the goal-event $G_i\langle\varphi_i\rangle$.

Consider the example BDI goal-plan hierarchy of Figure 3. The structure has just a single event-goal type $G\langle\rangle$ and three options to handle it, one of which (P_2) in turn posts two instances of the same event-goal type $G\langle\rangle$. In this way, the only plans that take an action in the environment are P_1 and

P_3. The figure highlights an execution trace as follows:

$$\lambda = G\langle\varphi_1\rangle[P_2 : w_1]{\cdot}G\langle\varphi_2\rangle[P_1 : w_1]{\cdot}G\langle\varphi_3\rangle[P_2 : w_2]{\cdot}G\langle\varphi_6\rangle[P_1 : w_2]{\cdot}G\langle\varphi_7\rangle[P_3 : w_3].$$

The first choice in the execution results in the selection of plan P_2 to handle event-goal instance $G\langle\varphi_1\rangle$ in a given world w_1. Plan P_2 in turn immediately posts the event-goal instance $G\langle\varphi_2\rangle$ that is successfully handled by the non-recursive node P_1. Plan P_2 then posts the second event-goal instance $G\langle\varphi_3\rangle$, which then is handled by itself in a recursive manner. The outcome is that λ traces a path that involves the successive execution of leaf plan P_1 for event-goal $G\langle\varphi_2\rangle$ followed by another execution of P_1 for event-goal $G\langle\varphi_6\rangle$, finally terminating in the failure of leaf plan P_3 for event-goal $G\langle\varphi_7\rangle$. Note that if plan P_2 had instead been selected to handle $G\langle\varphi_7\rangle$ then a deeper recursive call would have ensued. Similarly if earlier in the execution trace plan P_2 was selected to handle event-goal $G\langle\varphi_2\rangle$ then a different recursive sub-tree (shown in Figure 3 as dotted nodes under $G\langle\varphi_2\rangle$) would have unfolded.

The immediate implication of a recursive goal-plan structure is that the size of the hierarchy is no longer static but instead unfolds in a dynamic manner. The issue stems from the fact that the recursion is *unbounded* because the context conditions that cause the recursion to terminate are initially unknown. So in order to know the context conditions we must recursively explore, but in doing so we risk infinite recursion because the context conditions that ought to terminate the recursion are unknown. This means that we may never find the "bottom" or leaf nodes. This has implications for any *bottom-up* strategies. For instance, our conservative recording approach of [14] and the coverage-based confidence measure of [15] both suffer from this problem. Incidentally, the simpler aggressive recording approach is not impacted by recursion as it does not consider the goal-plan structure.

One way to resolve this issue is to treat all recursive goals simply as subtrees in a static structure and limit the recursive *unfolding* to a maximum allowed depth. In this study we use this bounded recursion approach for handling recursive structures. It follows then that wherever a recursive structure applies, a maximum recursion value must always be supplied. This may not be an unrealistic requirement given that the domain expert will usually have some idea of how much recursion is sufficient for an event-goal type.

3.3. Calculating Confidence in the Decision Tree Classification

The typical use of decision trees lies in the *offline* induction from a complete training set. In that sense, the use of decision trees in our framework is unorthodox since the training set is built incrementally by recording samples after each new execution. This results in the training set being incomplete[4] in the early stages of learning leading to misclassification. A confidence measure in the decision tree classification is therefore desirable to address this issue. Previously in [15] we showed how the *coverage* of possible execution paths below the plan in the goal-plan hierarchy may be used to build such a measure. Here we propose a new confidence measure that still builds on the coverage idea but that does not suffer from its limitations (Section 2.2).

Our requirement for the confidence measure is that it be a monotonic function whose values transition from no confidence (0.0) to full confidence (1.0) based on experience. Specifically, the experiences we are interested in should constitute coverage of the plan complexity (number of sub-plan choices) and the domain complexity (number of world states in which the plan applies). Since an exact calculation of such coverage does not scale for all practical purposes then we are interested in an *approximate* coverage that is still representative of the state of affairs but is simpler to compute.

One way to achieve this is to use a monotonic decay function[5] (for instance $\epsilon_i = \epsilon_{i-i} * \delta$ where $\delta < 1.0$) but where the decay factor δ is tied to the complexity involved. This way, a plan that has a larger number of sub-plan choices will utilise a slower decay factor δ taking longer to reach full confidence $(1 - \epsilon)$ than another plan that has less choices to make. For goal-plan complexity this decay δ_{Pt} may be calculated offline by analysing the goal-plan hierarchy. A similar treatment is possible for domain complexity although the decay factor in this case cannot be pre-determined since the number of world states is not known upfront and is dependent on the domain. For domain complexity then, it may be reasonable to treat the decay factor δ_{Pd} as a parameter specified by the domain expert.

$$c_P = (1.0 - \epsilon_{Pt}) * (1.0 - \epsilon_{Pd}). \tag{1}$$

Equation 1 shows how the final confidence c_P is calculated for a given plan P. Here ϵ_{Pt} is the plans tree complexity decay while ϵ_{Pd} is the plans domain complexity decay. The actual updates to the decay values are performed

[4]Training data is incomplete in the sense that the agent has only collected a portion of the full data set required to learn the correct classification.

[5]This technique is frequently applied in machine learning algorithms for balancing between exploration of choices and exploitation of learning.

each time the plan P is executed while the rate of decay is governed by the decay factors δ_{Pt} and δ_{Pd} accordingly.

3.4. Handling Event-Goal Types

Our BDI learning framework account as presented earlier in [14] and [15] did not account for event-goal types. In practical BDI systems, it is often the case that a single plan will handle all instances of an *event-goal type*. Furthermore event-goal instance parameters will generally be included in the context logical formula. Recall our previous definition (Section 3) of the learning task. The simplifying assumption in this definition is that the event-goal in question is an *event-goal instance*.

Consider again the goal-plan structure in Figure 3 and the highlighted solution path terminating in the leaf plans indicated by the $\surd$ symbol. An important point here is that the indicated solution applies to the event-goal *instance* $G\langle\varphi_1\rangle$ and to that instance alone. For a different instance $G\langle\varrho_1\rangle$ the solution path would likely be different (one way to visualise this in the Figure 3 is to think of it as an animation where the event-goal parameters and the placement of the $\surd$ symbols changes on each frame). This means that event-goal instance parameters must also be considered as input for a plans decision tree in order to learn solutions per event-goal instance.

We include such an account by augmenting the training samples for the decision tree with the event-goal parameters. As such, the training set now contains samples of the form $[w \cup \varphi, o]$ where the world state w is the initial set of all relevant attributes that represent the state of affairs, φ is the set of all event-type parameters, and o is the outcome class (success or failure). Incorporating the event-goal parameter set φ in the training data is sufficient for learning with event-goal types, and no fundamental change to the framework is required.

3.5. Calculating Plan Selection Weights based on Confidence

Typical BDI platforms offer several mechanisms for plan selection from a set of applicable plans, such as plan precedence and meta-level reasoning. However, since these mechanisms are pre-programmed and do not take into account the experience of the agent, we provide a new *probabilistic plan selection* function for this purpose.

For each plan, given its expectation of success (as determined by its decision tree learning) and a confidence measure in this expectation (based on coverage), we calculate a final *selection weight* that is indicative of the likelihood of the plan being selected for execution. Equation 2 shows how the plan selection weight $\Omega_P(w)$ is calculated for a given world state w. Initially, the confidence c_P is zero and the weight takes the default value of 0.5. Over time, as the confidence improves towards the final value of 1.0, the selection weight approaches the value $\kappa_P(w)$ estimated by the plan's decision tree.

$$\Omega_P(w) = 0.5 + [c_P * (\kappa_P(w) - 0.5)] . \tag{2}$$

Given the set of applicable plans for resolving event-goal G in world state w then, our probabilistic plan selection mechanism chooses a plan P_i with a probability directly proportional to its selection weight $\Omega_{P_i}(w)$. Such selection ensures a balance between the *exploitation* of current know-how and the *exploration* of new choices that is necessary for online learning tasks.

4. A Case Example: The Hanoi Towers Robot

To evaluate our learning framework we consider an existing BDI program from the JACK agent platform distribution [10]. The example involves a robot playing the well-known Towers of Hanoi game where the goal is to stack discs of decreasing size onto a single pin. The rules of the game forbid discs to be moved onto smaller discs, however top discs may be moved onto discs of larger size across three pins. The problem is interesting for our purposes since the example solution makes use of event-goal types and recursion. Furthermore, unlike our previous evaluations [14, 15] with synthetic plan libraries, here the evaluation criteria is clear: *does our learning framework achieve the performance of the existing system?*

The example solution consists of a `Player` agent that solves the game for any given legal initial configuration. The game solving strategy is encoded in plan `DiscStacker` that solves for one disc at a time starting from the largest and ending with the smallest onto a chosen pin. This in turn is achieved by posting event `Solve(?d,?p)` for solving disc `?d` onto pin `?p`. There are four plans that are relevant for this purpose:

`SolveRight` This plan solves moving a disc to the pin it is already on. Since the goal is already true, the plan does *nothing*.

`SolveTopMove` This plan moves the disc `?d` to the destination pin `?p` if the disc is not already there and if the move is legal. The actual move

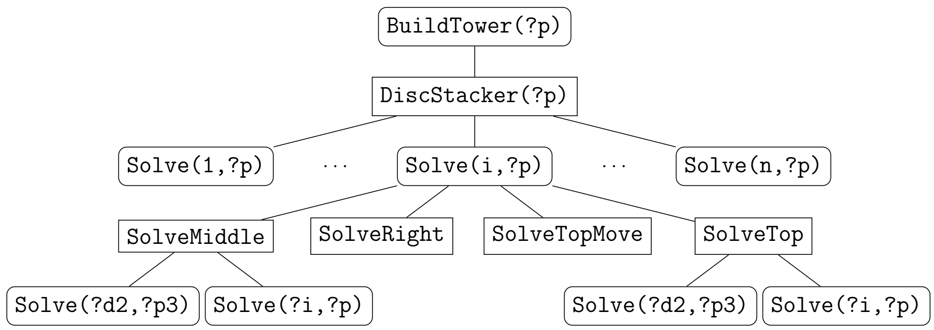


Figure 4: Goal-plan hierarchy for the Towers of Hanoi domain.

is perfomed by the primitive action `move(?p2,?p)`, where `?p2` is the source pin of disc `d`.

`SolveTop` This plan solves for the case when the disc `?d` may be legally lifted but cannot be legally placed at the destination because the top disc on the destination pin is smaller than `?d`. In this case, the plan first moves all the discs in the destination pin that are smaller than disc `?d` to the third (auxiliary) pin, and then re-posts the sub-goal to move `?d` to pin `?p` i.e. `Solve(?d,?p)`.

`SolveMiddle` This plan solves moving a disc from the *middle* of a stack. In this case, the plan first clears the source pin so that disc `?d` becomes the new top of the pin. This is done by solving for sub-goal `Solve(?d2,?p2)` where disc `?d2` is the disc currently on top of `?d` and `?p2` is the auxiliary. Subsequently the plan re-posts the sub-goal of moving `?d` to pin `?p` i.e. event `Solve(?d,?p)`.

Figure 4 illustrates the goal-plan hierarchy for the domain. Here we focus on learning the recursive typed `Solve(?d,?p)` event for which we remove the context conditions from the example plans and apply our framework.

4.1. Experimental Setup

The aim of this study is to evaluate our learning framework for recursive event-types. For this reason our experimentation with the Hanoi problem focusses on learning to resolve the recursive event `Solve` only and not on learning the strategy that solves the full Hanoi towers problem (this is done by `DiscStacker(?p)`). Since the full set of possible `Solve` events and initial pin configurations is large, our first step is to construct a sufficiently rich subset that we will use to evaluate our learning approaches.

We proceed by running the original Hanoi program for a number of randomly generated `Solve` events. For each run we record the `Solve` event, the initial pins configuration, and the maximum recursion encountered for the solution. This gives us a bag of several initial configurations for each recursion level that is a subset of all possible configurations.

Next, we run each candidate approach on the set of saved configurations for a given recursion level. i.e. where all solutions lie exactly at the specified recursion number. We use a fixed random generation seed for each experiment so that the same sequence of `Solve` events is generated for each learning approach. This isolates any environmental factors and allows us to attribute any differences in performance to the learning approaches alone.

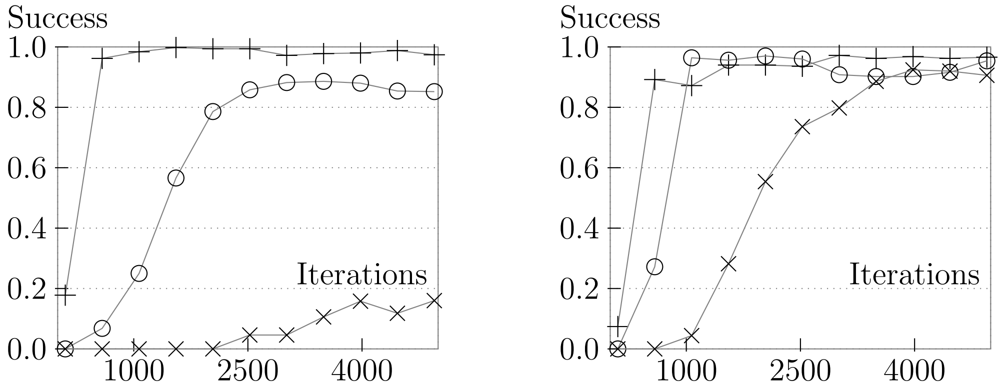
Finally, since in the Hanoi case we do not care about continuing exploration once a solution is found (optimality of solution is not a requirement), we have implemented two optimisations to the plan selection that boost the selection of plans with known solutions. Firstly, for a given plan, the confidence calculation of Equation 1 is performed only when no solution has previously been found for the given world state, otherwise full confidence is temporarily assumed for that plan. In addition, the plan selection mechanism of Equation 2 uses confidence only when no plan has a high expectation of success ($\kappa_P(w) > 0.95$) with high confidence ($c_P(w) > 0.95$), otherwise full confidence is temporarily assumed for *all* plans.

4.2. Results

The following results are for a Hanoi problem with $five$ discs[6]. Each plans domain complexity decay factor for the confidence calculation of Equation 1 are set to $\delta_{Pd} = 0.9$. For goal-plan tree complexity we use $\delta_{Pt} = 1$ for non-recursive plans and $\delta_{Pt} = \left[1 - (1/r^k)\right]$ for recursive plans where r is the run-time recursion level and k is a scaling factor. Currently, both δ_{Pd} and k are arbitrarily selected for the domain[7]. In all experiments, the recursion is bound to a maximum of *eight* levels that is sufficient to solve all configurations for a five-disc Hanoi problem. The performance of two learning

[6]We use five discs in order to keep the state space rich enough yet sufficiently small to allow learning runs to be completed and evaluated in reasonable time.

[7]In future work we hope to establish principles for determining general parameters.

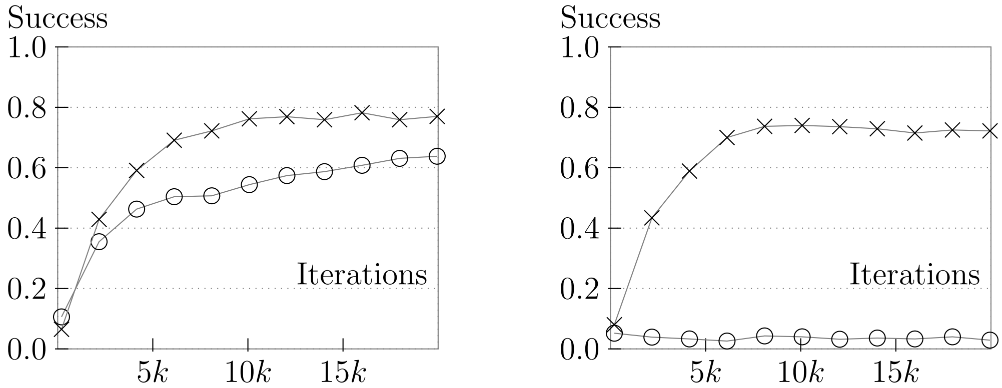


(a) ACL (b) ACL+Ω

Figure 5: Agent performance under ACL and ACL+Ω schemes for solutions at recursion levels one (pluses), three (circles) and five (crosses). Each point represents results from 5 experiment runs using an averaging window of 100 samples.

configurations is contrasted. The baseline learning algorithm ACL refers to the original aggressive learning approach of [14] and [15] using the original probabilistic plan selection function that has no confidence-based bias (uses decision tree expectation of success only). The new algorithm is referred to as ACL+Ω and uses the same aggressive learning approach as the former but combined with the new confidence-based probabilistic selection function (Equation 2) presented in this study.

To understand how the two approaches perform for solutions of varying difficulty we conducted a set of experiments with solutions at various recursive levels. Each experiment consisted of learning to resolve a given set of `Solve` events saved earlier as explained in Section 4.1 and whose solutions all required the same recursive depth. Figure 5 shows the results for ACL and ACL+Ω for solutions at one, three and five recursion levels respectively. Figure 5(a) shows that as the solution difficulty increases, ACL performance drops. For instance, for solutions at recursion level five, ACL is only able to resolve 20% of the solutions by the end of the experiment. This is expected because as the solutions become harder and require more number of `move(?p2,?p)` steps, the probabilistic plan selection of ACL that only considers the decision tree expectation of success (that is equally zero for all plans because no solution has been found yet) has progressively less chances



(a) No Applicability Threshold

(b) Applicability Threshold 20%

Figure 6: Agent performance under ACL (circles) and ACL+Ω (crosses) schemes with applicability thresholds. Each point represents results from 5 experiment runs using an averaging window of 200 samples.

of exploring to deeper levels. The confidence-based measure of Equation 2 however, that takes into account the goal-plan tree complexity, is able to guide the exploration towards the deeper solutions. This is evident in Figure 5(b) where ACL+Ω finds all level five solutions by experiment end.

Next, we conducted an experiment that consisted of resolving the full set of saved `Solve` events i.e. the set had solutions dispersed through all possible recursion levels. Figure 6(a) shows the results for the two approaches. Overall we expect ACL+Ω to do better than ACL which is the case although the performance benefits of the confidence-based selection are less obvious since approximately 75% of the solutions require four recursion levels or less which both approaches perform reasonably at.

An important consideration during plan selection is the issue of applicability thresholds. In the classical BDI framework plans are either applicable or not in a boolean decision. However, in our modified framework plans are applicable according to the selection weight given by Equation 2. Since plan execution is often not cost-free in real systems, it is likely that an adequate plan selection scheme would not select *any* plan if they all have too low an expectation of success. In fact, an agent may then fail a plan without even trying if the cost of trying and possibly failing is considered too high.

To represent this scenario we run the earlier experiment but using an applicability threshold of 20% whereby plans with expectations of success less than this threshold would not be selected. Figure 6(b) shows the learning outcome for the two approaches in this case. As reported in [15] earlier and confirmed here, ACL shows a complete inability to learn in this case. ACL+Ω on the other hand benefits from the adjusted selection weights of Equation 2 and is not impacted by the threshold constraint[8].

A final observation in Figure 6 is that neither approach converges to 1.0. This is because the Hanoi domain is not free from goal inter-dependence. So if a plan learns to solve a sub-goal in two ways that each lead to different end states, then depending on the way the sub-goal was resolved the top level goal may pass or fail. Indeed, when we analyse the result log we find that ACL+Ω finds all solutions as expected, but even so fails now and then for some of these because of the way a sub-goal was realised. Our learning framework does not currently support goal inter-dependence.

5. Discussion and Conclusion

This paper builds on earlier work that extends the typical BDI programming framework to use decision trees as (part of) a plan's context condition, with a probabilistic plan selection mechanism that caters for both exploration and exploitation of plans. We have shown in earlier work that due to the structure of BDI programs, care must be taken in how learning is used, to avoid problems in certain situations. In some cases these problems lead to failure to learn at all, as we also show here.

In this paper we extend previous work to allow for parameterised goals such as *Travel($from, $to)* and also for recursion, both of which are necessary for real applications. In doing this our previous confidence measure which relied on a finite goal-plan tree did not scale, so we have provided a more approximate measure, relying on the principles that have been shown correct, but without the limitations. This paper also takes an existing BDI program involving parameterised goals and recursion, and evaluates our approach using this program. By removing the existing context conditions, and then learning the correct behaviour, we show that we are able to obtain good

[8]Note that in the general sense it is perfectly possible that the Equation 2 weight will fall below any given threshold. This is the case when no solution has been found by the time full confidence is reached according to Equation 1.

(although not perfect[9]) performance. We also demonstrate that the naive approach to learning, that does not account for the BDI program structure fails to learn given some program structures and an applicability threshold.

The work still contains a number of simplifying assumptions which will need to be addressed before being able to develop *practical* on-line learning for situated agents. Currently the work has not been integrated with standard BDI failure handling and recovery. Clearly this will be needed, but we do not expect that it will undermine any of our results described here. In fact a careful integration of failure handling could improve the speed of learning as multiple attempts could be made to achieve a (sub)-goal. However care will need to be taken regarding changes to world state and possible interactions between failed attempts and eventually successful ones.

Our current approach does not detect and learn interactions between sibling goals in the context of a particular parent; each subgoal is treated "locally." To handle such interactions, the selection of a plan for resolving a sub-goal should also be predicated on the goals higher than the sub-goal, that is, it should take into account the "reasons" for the sub-goal.

Finally, we note that our current use of decision trees is naive. For instance, in our framework all execution data is maintained forever and decision trees re-built after each plan execution. Furthermore, we learn using actual world states, whereas an improvement would be to learn using relational world information. While not ideal, our approach nevertheless allows us to focus on understanding the complexities of learning in BDI programs first without worrying about the underlying techniques.

The issue of combining learning and deliberative approaches for online decision making in BDI-like systems has not been widely addressed. In [18], *preiously learnt* low level robot soccer skills are used in the deliberative decision making process once deployed. Hernández et al. [19] give a preliminary account of how decision trees may be induced on plan failures in order to find alternative logical context conditions in a deterministic paint-world example. A closely related area to BDI is that of hierarchical task network (HTN) systems where task decompositions used are similar to BDI goal-plan hierarchies. Recently, in similarly motivated work to ours, [20] proposed a method for learning HTN method preconditions under partial observations.

[9] We would hope that when learning is combined with programmer provided context conditions, the problems preventing perfect learning here would be avoided.

There, a set of constraints are constructed from observed decomposition trees that are then solved *offline* using a constraint solver. In contrast, in our work learning and deliberation are fully integrated in a way that one impacts the other and the classical exploration/exploitation dilemma applies.

Although there is still work to do before we can expect learning to be successfully integrated into a fully autonomous BDI agent, the work reported here is significant in that it provides a solid foundation for adding new capabilities to BDI agents to allow them to learn and adapt based on experience.

[1] N. R. Jennings, An agent-based approch for building complex software systems, Communications of the ACM 44 (4) (2001) 35–41.

[2] R. A. Belecheanu, S. Munroe, M. Luck, T. Payne, T. Miller, P. McBurney, M. Pechoucek, Commercial applications of agents: Lessons, experiences and challenges, in: Proceedings of Autonomous Agents and Multi-Agent Systems (AAMAS), ACM Press, Hakodate, Japan, 2006, pp. 1549–1555.

[3] M. Ljungberg, A. Lucas, The OASIS air-traffic management system, in: Proceedings of the Pacific Rim International Conference on Artificial Intelligence (PRICAI), Seoul, Korea, 1992.

[4] Z. Tu, M. Ferry, G. Prickett, C. Heinze, Truly autonomous UAVs and teaming, in: Twelfth Australian Aeronautical Conference, 2007, pp. 483–489, engineers Australia.

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

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

[7] M. Bratman, Intentions, Plans, and Practical Reason, Harvard University Press, 1987.

[8] D. C. Dennet, The intentional Stance, MIT Press, 1987.

[9] S. S. Benfield, J. Hendrickson, D. Galanti, Making a strong business case for multiagent technology, in: Proceedings of Autonomous Agents and Multi-Agent Systems (AAMAS), ACM Press, 2006, pp. 10–15.

[10] P. Busetta, R. Rönnquist, A. Hodgson, A. Lucas, JACK intelligent agents: Components for intelligent agents in Java, AgentLink Newsletter 2 (1999) 2–5, Agent Oriented Software.

[11] A. Pokahr, L. Braubach, W. Lamersdorf, JADEX: Implementing a BDI-infrastructure for JADE agents, EXP - in search of innovation (Special Issue on JADE) 3 (3) (2003) 76–85.

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

[13] T. Mitchell, Machine Learning, McGraw Hill, 1997.

[14] S. Airiau, L. Padham, S. Sardina, S. Sen, Enhancing adaptation in BDI agents using learning techniques, International Journal of Agent Technologies and Systems (IJATS) 1 (2) (2009) 1–18.

[15] D. Singh, S. Sardina, L. Padham, S. Airiau, Learning context conditions for BDI plan selection, in review process (2009).

[16] M. P. Georgeff, F. F. Ingrand, Decision making in an embedded reasoning system, in: Proceedings of IJCAI, Detroit, USA, 1989, pp. 972–978.

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

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

[19] A. Guerra-Hernández, A. E. Fallah-Seghrouchni, H. Soldano, Learning in BDI Multi-agent Systems, Vol. 3259 of LNCS, Springer, 2004, pp. 218–233.

[20] H. Zhuo, D. Hu, C. Hogg, Q. Yang, H. Munoz-Avila, Learning HTN Method Preconditions and Action Models from partial Observations, in: Proceedings of the Twenty-First International Joint Conference on Artificial Intelligence (IJCAI-09), 2009.