Enhancing the Adaptation of BDI Agents Using Learning Techniques

Stéphane Airiau

ILLC - University of Amsterdam

Amsterdam, NL

stephane@illc.uva.nl

Lin Padgham

RMIT University

Melbourne, Australia

lin.padgham@rmit.edu.au

Sebastian Sardina

RMIT University

Melbourne, Australia

sebastian.sardina@rmit.edu.au

Sandip Sen

University of Tulsa

Tulsa, OK, USA

sandip@utulsa.edu

Abstract

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

.

INCORPORATING LEARNING IN BDI AGENTS

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 (Bratman, Israel, & Pollack, 1988) 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 in response 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, in that they are confined to what their pre-defined plan libraries encode. As a result, they cannot significantly alter their behaviors from the ones specified during their initial deployment. In particular, these agents are not able to *learn* new behaviors (i.e., new plans) nor to learn how to choose among plans—both plans and their contexts are hard-coded. In this work, we are concerned with the latter limitation. We therefore analyze BDI-based agent designs and identify opportunities and mechanisms for performing plan context learning in typical BDI-style agents. By doing so, agents can enjoy a higher degree of adaptability by allowing them to improve their plan selection *on the basis of analysis of experiences*.

Research in machine learning can be broadly categorized into *knowledge-rich* and *knowledge-lean* techniques. Whereas some researchers have proposed and investigated learning

mechanisms that incorporate and utilize significant amounts of domain knowledge (DeJong & Mooney, 1986, Ellman, 1989, Kolodner, 1993), the large majority of popular learning techniques assume very little domain knowledge and are largely data, rather than model, driven (Aha, Kibler, & Albert, 1991, Booker, Goldberg, & Holland, 1989, Kaelbling, Littman, & Moore, 1996, Krause, 1998, Quinlan, 1986, Rumelhart, Hinton, & Williams, 1986). Research in multiagent learning (Alonso, d'Inverno, Kudenko, Luck, & J.Noble, 2001, Panait & Luke, 2005, Tuyls & Nowé, 2006) has also followed this trend. This is particularly unfortunate as practical multiagent systems are meant to leverage existing domain knowledge in order to facilitate scalability, flexibility, and robustness. For most such online, real-time multiagent systems, individual agents need to quickly and effectively respond to unforeseen events as well as to gradual changes in environmental conditions. In this context, the amount of experience and adaptation time available will be orders of magnitude less than what is assumed by offline knowledge-lean learning algorithms. As a result, techniques that take advantage of the available domain knowledge to aid and guide the learning and adaptation process are key to the development of successful agent learning approaches.

So, in the context of BDI agents, we foresee significant synergistic possibilities for combining *learning* and *reasoning* mechanisms. Whereas available domain knowledge of BDI agents can inform and direct embedded learning modules, the latter can incrementally adapt and update components of the reasoning module to "tune" the agents' behaviors. It may well be the case that while substantial knowledge is encoded at design-time, there are additional *nuances* which can be learnt over time that will eventually yield better overall performance. In this paper, then, we discuss issues and propose preliminary techniques in order to refine coarse heuristics

provided by the BDI programmer at design time for the purposes of doing plan selection, that is, we focus on mechanism for "improving" the *context conditions* of existing plans.

The rest of the paper is organized as follows. In the next section, we provide an overview of the relevant aspects of typical BDI-style agents. We then discuss modifications to the BDI framework so as to include mechanisms for improving plan selection by refining the context conditions of plans. We do so by relying on a number of simplifying assumptions that make the scenario an "ideal" one. After that, we outline ways in which these assumptions may be lifted. We end the paper by drawing conclusions and future lines of work.

BDI Agent-Oriented Programming

The BDI (Belief-Desire-Intention) model is a popular and well-studied architecture of agency for intelligent agents situated in complex and dynamic environments. The model has its roots in philosophy with Bratman's (Bratman, 1987) theory of practical reasoning and Dennett's theory of intentional systems (Dennett, 1987). BDI agent-oriented systems are extremely flexible and responsive to the environment, and as a result, well suited for complex applications with (soft) real-time reasoning and control requirements. There are a number of agent programming languages and development platforms in the BDI tradition, such as PRS (Georgeff & Ingrand, 1989), JAM (Huber, 1999), JACK (Busetta, Rönnquist, Hodgson, & Lucas, 1998), 3APL (Hindriks, Boer, Hoek, & Meyer, 1999), Jason (Bordini, Hübner, & Wooldridge, 2007).

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 form $e{:}\psi \leftarrow P$

encoding the standard domain operation-strategy *P* (that is, a program) for handling an event-goal *e* when context condition ψ is believed to hold. Importantly, new *internal* events may be "posted" during the actual execution of program *P*. 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.

In some way or another, all the above agent programming languages and development environments capture the basic reactive goal-oriented behavior: a BDI system responds to events, the inputs to the system, by committing to handle one pending event-goal, selecting a plan rule from the library, and placing its plan-body program into the intention base. More concretely, these system generally follow Rao and Georgeff's abstract interpreter for rational agents (Rao & Georgeff, 1992) which, roughly speaking, is repeats the following steps (illustrated in Figure 1):

1. Select a *pending*—internal or external—*event* (if any).
2. *Update* beliefs, goals, and intentions—new information may cause the agent to update its beliefs and modify its goals and intentions.
3. Using the plan library, determine the set of *applicable plans* to respond to the selected event, that is, identify those plans that are relevant to the event type and whose context condition are believed true.
4. *Choose one applicable plan* and push it into the intention structure, either as a completely new intention for an external event or by expanding the details of the intention that produced the internal event.
5. Select one intention and *execute* one or more steps on it. This execution may, in turn, generate new *internal* events.

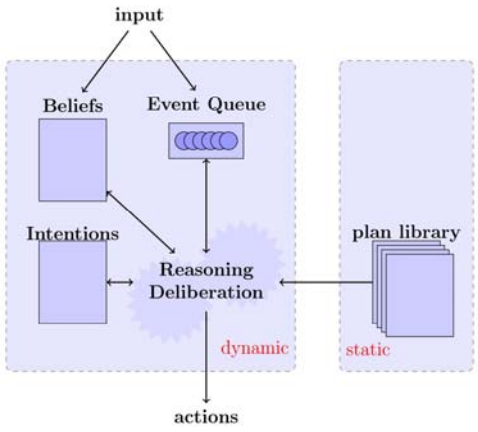

Figure 1: Architecture of a typical BDI Agent System

For the purposes of this paper, we shall mostly focus on the plan library, which contains the procedural domain knowledge of the domain by encoding the typical operations. Intuitively, a plan of the form $e{:}\psi{\leftarrow}P$ is meant to state that program P is a reasonable strategy to follow in order to address the event-goal e whenever ψ is believed to hold. Event e is referred as the trigger of the plan, ψ its context condition, and program P the plan's body. A plan body typically contains actions (*act*) to be performed in the environment, belief updates ($+b$ and $-b$ to add or delete a proposition, respectively) and tests ($?\ \phi$) operations, and subgoals ($!e$) to post internal events which ought to be in turn resolved by selecting suitable plans for that event.

By grouping together plans which respond to the same event type, the plan library can be seen as a set of *goal-plan tree* templates, where goal (or event) nodes have children representing the alternative plans for achieving the goal, and in turn, plan nodes have children nodes representing the subgoals (or actions) of the plan . These structures, depicted in Figure 2, can be seen as "AND"/"OR" trees: for a plan to succeed all the subgoals and actions of the plan must be successful ("AND"); for a subgoal to succeed one of the plans to achieve it must succeed ("OR").

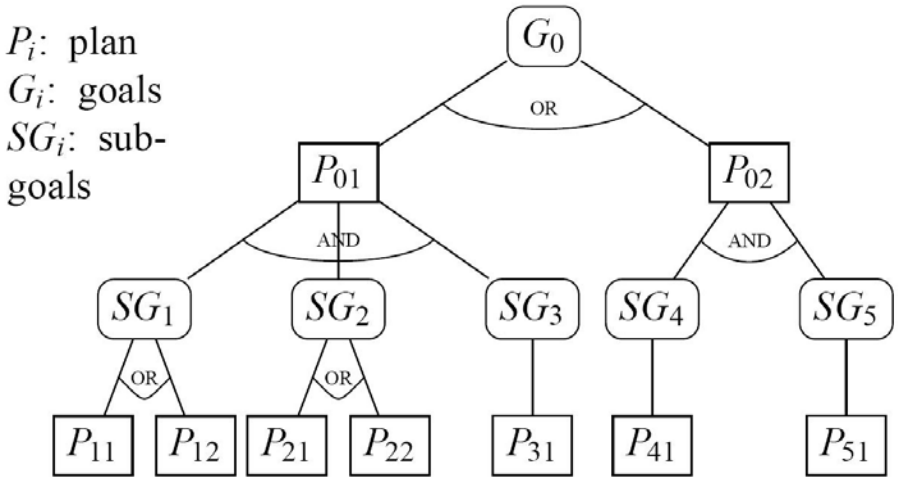

Figure 2: Goal-plan tree hierarchy

When a step in a plan happens to fail (e.g., an action cannot be executed, a test condition does not apply, or subgoal itself fails), this causes the whole plan to fail, and an alternative applicable plan for its parent goal is tried. If there are no alternative applicable plans, the parent goal itself fails, cascading the failure and search for alternative plans upwards one level in the goal-plan tree. It is exactly this context-sensitive search of alternative plans for a goal that enables BDI systems to robustly recover from failure situations, particularly, those where the environment has changed in an unexpected manner.

Interestingly, the structured information contained in the goal plan tree can also be used to provide guidance to the learning module. In particular, consider the context condition of plans, which are critical for guiding the execution of the agent program. A plan will not be used in the current state if its context condition is not satisfied. Incorrect or inadequate context conditions can lead to two types of problems. If the context condition of a plan is *over-constrained* and evaluates as false in some situations where the plan could succeed then this plan will simply never be tried in those situations, resulting in possible utility loss to the agent. On the other hand,

if the context condition is *under-specified*, it may evaluate to true in some situations where the plan will be ineffective. Such "false triggers" will result in unnecessary failures, and although the agent may recover by choosing alternative plans, it may lose valuable time or waste resources unnecessarily, thereby losing utility. Hence, it would be preferable to learn from experience to avoid using plans that are unlikely to succeed at particular environmental states.

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

Refining the Context Conditions of Plans

We start by considering an idealized setting and explore ways in which plan selection could be refined based on experience. The "ideal" setting allows us to describe and understand the ideal situation, noting that non-trivial reasoning is already required if we are to eventually exclude all unnecessary failures.

The most straightforward way of refining plan selection is to gradually modify the context conditions of plans so as to make them more specific. We shall also briefly discuss a more subtle refinement that could be done by a smarter reasoner, regarding selection of plan *sequences* in specific situations.

Assumptions

In designing the idealized setting, we shall make a number of simplifying assumptions, namely:

a) The environment in which the agent is situated is assumed to be *deterministic* and *fully observable* by the agent. Thus action actions outcomes (including its success or failure) are

always the same for each particular state and the agent always knows in which state she is in. This shall allow us to always make *correct* updates to context conditions.

b) The initial plans' context conditions provided at design-time are understood as *necessary conditions*, that is, as minimum constraints for the plan to succeed—our goal is not to repair "erroneous" context conditions, but to further refine them. This assumption amounts to the programmer encoding the knowledge she is *certain* about in the context condition of the agent, with the hope that the learning mechanism will complete the unknown "gaps."

c) Plan bodies are restricted to sequences of actions and subgoals, and intentions are executed in a linear manner, that is, without any interleaving. This removes the need to monitor for interactions between goals that could have an effect on success. Interleaving can be allowed, provided a suitable reasoning module can guarantee the linearization-equivalence of their executions with respect to the goals pursued.

d) Typical BDI failure recovery, under which different available options for a goal are tried when a plan ends up failing, is assumed *disabled*. Learning in the context of BDI failure recovery poses new challenges and issues that we have not considered at this point.

Besides these core restrictions, we should also make a number of technical assumptions:

- Beliefs and context conditions of plans are expressed in propositional logic. Let V denote the set of available propositions in the domain.
- The agent has a complete and correct axiomatization of the dynamics of the domain. This means that all changes to any variable $v \in V$ are explicit to the agent. This allows us to reason about changes caused by the agent, as opposed to those happening independently in the environment.

- A superset of all propositional variables that are (potentially) *relevant* for the success/failure of a plan *P* is known and defined at design-time. We shall refer to that set as R_P (of course, $R_P \subseteq V$). The idea is that although only some of the variables in R_P may be used to express the initial context condition of *P*, no variable outside such set may influence the success or failure of *P*. (A likely candidate for R_P can be identified as the set of variables that are accessed by some plan below the parent goal of *P* in the goal-plan tree.)

When *s* is a complete state of the world, we denote with $s[R_P]$, the tuple containing the values of propositions in R_P in state *s*.

Refining Context Conditions: The Ideal Case

We will now discuss the situations in which one can refine a context condition (to avoid failure due to poor plan choice) in a provably correct way.

Recall that the context condition of a plan indicates the situations in which the plan is expected to succeed, provided the right plan decisions are made for the sub-goals posted in the plan. Thus, for a BDI agent to be efficient, it is important that plans have accurate context conditions so that the right choices are made. There are two possible scenarios explaining a failure of a plan *P* in a state *s*:

a) There is no path in *P*'s sub-tree that leads to a success of *P*. This means that the choice to use *P* under state *s* was not correct. The challenge here is to know whether everything in *P*'s sub-tree has been attempted (without success).

b) The choice of a particular action or sub-goal within *P*'s sub-tree was not correct. For example, suppose that within *P*'s sub goal-plan tree, the agent faces a choice between two sub-plans, and only one is bound to succeed. When the agent decides to use the wrong sub-plan, *P* may also end up failing (this is exactly what BDI failure recovery would often help to

address. However, in this work, we assume that BDI failure retry mechanism is "disabled."). In that case, the initial decision of using *P* was indeed correct, but a later (wrong) choice provoked the failure.

So, let us consider the example depicted by the plan-goal tree in Figure 2. Assume that (deterministic) plans P_{11}, $P_{12,}$ $P_{21,}$ P_{22}, and P_{31} each consists of a sequence of one or more atomic actions and suppose that plan P_{01} was chosen for execution at a state *s*. Suppose next that plan P_{11} is started in some state *s* to address subgoal SG_1, and that, after some steps, plan P_{11} happens to fail. Such failure will in turn cause the failure of goal SG_1 itself and of plan P_{01} as well (remember we assume no failure recovery so plan P_{12} would not be tried upon failure). The question then is: *when can we update the context condition of plan* P_{01} *to exclude state* $s[\mathsf{R}_{P_{01}}]$*?*

Observe that one cannot directly rule out (sub)state $s[\mathsf{R}_{P_{01}}]$ from P_{01}'s context condition as soon as P_{01} fails, since P_{01}'s failure could have been avoided had plan P_{12} been chosen instead for achieving SG_1. However, if we have previously recorded that option P_{12} has already failed in the past when started in a state *s*', such that $s'[\mathsf{R}_{P_{12}}] = s[\mathsf{R}_{P_{12}}]$, then we would be justified in updating the context condition of P_{01} to disallow (sub)state $s[\mathsf{R}_{P_{01}}]$. This is because *all* possible execution paths of P_{01} from state $s[\mathsf{R}_{P_{01}}]$ are bound to fail. Notice that this reasoning is valid only the assumptions that the domain is deterministic and fully-observable, and that R_P includes all the propositions that could influence, directly or indirectly, the success of plan *P*.

Let us now slightly modify the situation and assume the failure occurs in plan P_{31} instead, thus propagating upwards to P_{01}. In this case, it is correct to further constrain P_{01}'s context condition (to rule out substate $s[\mathsf{R}_{P_{01}}]$) *only if* all combinations of plan choices for previous subgoals SG_1 and SG_2 have been tried (in states compatible with $s[\mathsf{R}_{P_{01}}]$). Otherwise, plan P_{31} may have failed only due to an "incorrect" combination of plans for goals SG_1 and SG_2.

Nevertheless, there is some additional selection information that an agent could potentially use in such cases. For example, suppose that the failure of P_{31} occurred after having executed plans P_{11} and P_{21} to achieve SG_1 and SG_2, respectively. Then, one can conclude that, in the context of plan P_{01} and in states compatible with $s[\mathsf{R}_{P_{01}}]$, the combination of plans P_{11} and P_{21} is "bad" one, as it would eventually cause P_{31} to fail. In future cases, hence, the agent should avoid such combination of plan choices for achieving SG_1 and SG_2. Interestingly, however, this is *not* information that could be represented as part of the context condition of plans. The reason is that context conditions are *local* to their particular plans and do not take into account the various ways may be combined (in higher-level plans) to achieve different goals. Instead, one should see this knowledge as meta-level information that could be used by a (complex) plan selection function in order to choose (or exclude) particular paths through the goal-plan tree.

We close this section by pointing out an interesting application of *automated planning* in this context. In principle, the goal-plan tree has been initially designed so that if the context condition of a plan is met, there is indeed a path leading to a success of the plan. By refining the context condition of a plan (e.g., plan P_{31}), the agent could eventually rule out any successful execution for a plan higher up in the hierarchy (e.g., plan P_{01}). In such a case, it would be justified to also modify the higher level plan's context condition (e.g., the context of plan P_{01}). So, in general, once we have detected a failure and modified the corresponding failed plan's context condition (e.g., plan P_{31}'s context), we may want to propagate context modifications as far up the goal-plan tree as is justified. To that end, one could rely on automated planning, in particular, on *Hierarchical Task Network* (HTN) plan decomposition, as described in (Sardina, de Silva, & Padgham, 2006). The idea is to verify if, given the modifications done in lower-level plans, there is still a potential solution for a higher-level plan. If not, then we can update the

higher-level plan's context condition, and so on. In our example, if after modifying the context condition of P_{31}, we attempt offline HTN planning on plan P_{01} from state s and fail to find a decomposition solution, then we can modify the context condition of P_{01} to exclude substate $s[\mathsf{R}_{P_{01}}]$: by offline reasoning we have already discovered that there is no valid decomposition for P_{01} in s.

Refinement of Context Conditions in Nondeterministic Environments

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

Decision Trees (Mitchell, 1997) are a natural learning mechanism choice for this situation, since they can deal with "noise" (created by the non-determinism in our case) and they are able to support disjunctive hypotheses. What is more, a decision tree is readily convertible to *rules*, which are the usual representation of context conditions. We associate then a decision tree with *each plan* in the goal-plan tree. As we build up and record experiences, the decision tree will identify the conditions under which success and failure are likely, allowing us to modify context conditions in a similar way to our simple example in the previous section.

Typically, a decision tree is built *offline* from a set of data. In our case, though, we would like the agent to act as best she can, while accumulating experience so as to make improved

decisions as more information is gathered. Algorithms for incremental induction of decision are available (Swere, Mulvaney, & Sillitoe, 2006, Utgoff, Berkman, & Clouse, 1997).

There are two important and inter-related questions to answer when using a decision tree to learn the context condition of a plan: *(a)* when is it justified to *record* data (in particular, failure data)? ; and *(b)* when and how the decision tree may be used for plan selection? The first issue involves deciding when a failure in a plan-node is meaningful for such node. The second one is related to plan applicability and actual plan selection, and how the current decision tree at hand can be used to determine whether a plan is applicable and which plan to choose among the applicable ones.

Recording data in a plan's decision tree

We have a decision tree linked to each plan, capturing the past experience of the agent in executing that plan. Upon execution of a plan, the agent must decide whether to incorporate the information about such execution into the corresponding plan's decision tree. This is particularly important if the plan happens to fail: Should the agent update the decision tree with failure information?

First of all, in a non-deterministic environment, there will inevitably be some "noise" in the data gathered and a plan *P* may fail where it generally works. One reason for plan *P* to fail is an (unexpected) change of some variable in R_P due to some environment factors outside the control of the agent. Such causes of failure should not lead us to conclude that plan *P* was a bad choice. Instead, one should monitor for and discard situations where the relevant variables for plan *P* have been affected outside of the agent's control, so such failures would not be recorded (this assumes there is no correlation between *P*'s execution and the unexpected changes in the environment). Alternatively, one could record them and assume that these data points will

eventually be treated as noise. For simplicity, in our empirical investigation we did not consider any external effects so that the agent is the only one performing actions in the environment.

Based on the analysis done in the previous section, one could argue that it does not make sense to record the (failure) data for a plan P when P fails due to a poor choices made within P's sub-tree. Again, if plan P_{01} would have worked well had plan P_{12} been chosen instead of P_{11}, then the failure of P_{01} is in a sense spurious, corresponds to a false negative, and should not be recorded in P_{01}'s corresponding decision tree. Once accurate selections are regularly being made below P_{01}, we expect the system to indeed take into account failures and update P_{01}'s decision tree.

Considering this, our initial approach was to delay the collection of data for a plan P until all plans in P's sub-tree are considered accurate or stable enough. We call such approach *bottom up learning* (BUL). Note that this poses another non-trivial challenge, namely, deciding when a decision tree for a plan is accurate/stable enough, that is, close enough to its "real" ideal context condition. In our empirical evaluation, we use the naiv*e* approach of requiring a minimum number of data points (*minNumRecords*). Algorithm 1 determines, recursively, when a decision tree is considered stable or accurate enough at a given plan-node. Clearly, this is an extremely simplistic way of evaluating the "quality" of a decision tree at hand in a plan-node and, in future work, we shall be exploring more sophisticated accounts.

Algorithm 1: *StableDecisionTree*(P)

```
/* numRecords(P) is the number of instances recorded for plan P's decision
   tree                                                                  */
if P is an action then
    // there is no sub-tree
    return (numRecords(P) > minNumRecords)
else
    if (numRecords(P) > minNumRecords) then
        for each subgoal G_p of P do
            for each plan Q satisfying G_p do
                if (not StableDecisionTree(Q)) then
                    return false;
        return true;
    else
        return false;
```

Inspired by the reasoning behind the "ideal" setting discussed in the previous section, the BUL bottom-up learning approach suffers from the drawback that it may in fact take a large number of execution instances before any data is collected for a plan *P*. This is particularly problematic in the context of multi-agent systems, in which we may not expect to execute the same plans a large number of times in short periods of time. Because of that, we also experimented with what we call the *concurrent learning* approach (CL), in which the agent always collects data at all nodes, regardless of the decision trees' qualities down the hierarchy. Obviously, the proportion of false negative instances is expected to be higher than when using the BUL method, but more data will be collected at each node regardless of its level within the hierarchy. If the plan selection mechanism promotes some exploration to compensate for those false negative instances (see below), it should still be possible to reach optimal behavior given enough instances.

The following Algorithm 2 shows the abstract procedure for deciding when to record an execution instance in a decision tree, depending on the strategy (BUL or CL) adopted.

Algorithm 2: *Record*(Plan P, State s, boolean out)

```
/* s: world state when P was selected                                  */
/* out: outcome of P's execution                                       */
/* AddInstance(P, s, o): add instance (s, o) to P's decision tree      */
switch selectionMode do
    case CL
        numRecords(P) + +;
        AddInstance(P, s, out);
    case BUL
        if StableDecisionTree(P) then
            numRecords(P) + +;
            AddInstance(P, s, out);
```

Using plans' decision trees for plan selection

The second issue to be addressed is the notion of plan applicability and plan selection in the context of the newly introduced decision trees. Typical BDI systems consider a plan applicable when its context condition ψ is believed true; and a plan is selected for execution among the applicable set. In our context, it is expected that, besides the initial context formula provided at design time, the decision tree attached to a plan also influences the applicability and selection of a plan.

So, given that we have assumed that the context condition ψ of a plan (set at design-time by the BDI programmer) is a necessary condition, the agent first filters plans with respect to their initial contexts. If the context condition of a plan does not hold, the plan is *not* considered as a candidate for selection. Once a plan has passed its initial context condition, the agent may want to consider also its corresponding decision tree, in order to exploit the experience obtained so far.

Thus, key issues regarding the use of the decision trees are *(a)* when to begin using the trees; and *(b)* how to allow for plan exploration in order to "correct" possibly spurious entries. These issues, together with that as to whether it is better to delay the data collection for higher

nodes until lower nodes are stabilized, are in fact the subject of our experimentation in the next section.

To address these issues we have considered the following three plan selection mechanisms that extend the standard BDI plan selection:

Simple Selection. In this plan selection approach, the agent uses the standard (random) plan selection among those plans considered applicable. However, in contrast with the standard BDI approach, a plan is considered applicable if its design-time context condition ψ holds and either its decision tree is not accurate/stable enough (under some account of stability; e.g., see Algorithm 1), or its decision tree is considered stable and provides a success rate higher than a given threshold (e.g., 50% success over past recorded instances).

Optimistic Selection. Under this account, the plan with the highest rate of success is selected. Such rate is obtained by computing the ratio between the number of positive and negative instances in the leaf nodes corresponding to the current state of the world in the relevant decision tree, provided such decision tree is considered stable enough. Otherwise, if the decision tree for a plan is not deemed stable yet, the highest rate is assigned to the plan. By doing that, we bias the plan selection to explore plans whose decision trees are not yet accurate enough so as to collect enough experience to bring them to a stable status.

Probabilistic Selection. Under this account, a plan is selected probabilistically according to the ratio of positive versus negative instances: the ratio between the number of positive and negative instances is computed as for the optimistic selection, and then, a plan is chosen with a probability *proportional* to the ratio of success (a minimal ratio is assigned to plans to avoid complete ruling out plans due to initial failures). In contrast with the two previous approaches, this method *always* takes into account the decision trees associated with plans, and as a result,

there is no need to check for stability of decision trees. This is because all plans—even those that have unfortunately failed in their first executions—would have some chance to be eventually selected for execution, and thus, plan exploration is always guaranteed.

From a conceptual perspective, wee believe that the probabilistic plan selection mechanism provides the most suitable account for our extended BDI agents, since it does not require any notion of decision tree "stability" and it provides a parsimonious exploration-exploitation scheme. This was later corroborated our experiments (see below). by In some sense, one can view this plan selection approach as generalizing the standard *binary* BDI notion of plan applicability to a *graded* account of applicability.

A high-level representation of these selection accounts is shown in Algorithm 3.

Algorithm 3: *Select*(State s, Goal g)

```
/* pass_context(PlanLibrary, g, s) selects the set of plans that satisfy goal g
   and which context condition is satisfied in world state s.                */
/* rnd_u(l): selects randomly item in the set l using a uniform probability
   distribution                                                              */
/* rnd_w(l): selects randomly an item in the set l of pairs (e, w) where the
   probability for selecting item e is proportional to w                     */
/* Prob_success(P, s): proportion of positive instances in the leaf node of P's
   decision tree for the state s                                             */
candidates ← pass_context(PlanLibrary, g, s) ;
applicable = ∅;
switch selectMode do
    case simple
        for all P ∈ candidates do
            if StableDecisionTree(P) then
                if (Prob_Success(P, s) > 0.5) then
                    applicable ← applicable ∪ {P}
            else
                applicable ← applicable ∪ {P} ;
        return rnd_u(applicable);
    case optimistic
        for all P ∈ candidates do
            if StableDecisionTree(P) then
                applicable ← applicable ∪ {(P, Prob_success(P, s))}
            else
                applicable ← applicable ∪ {(P, 1.0)} ;
        bestP ← argmax_(P,w)∈applicable w ;
        return rnd_u(bestP);
    case probabilistic
        for all P ∈ candidates do
            if StableDecisionTree(P) then
                applicable ← applicable ∪ {(P, Prob_success(P, s))}
            else
                applicable ← applicable ∪ {(P, 1.0)} ;
        return rnd_w(applicable);
```

Empirical Results

We conducted preliminary experiments to evaluate how agent performance is affected by the different issues discussed in previous sections. We concentrate on the following two issues:

a) Selecting a plan when multiple plans are applicable. We tested the three different selection mechanisms presented in the previous section, namely, simple selection, optimistic selection, and probabilistic selection.

b) Collecting data instances to build the decision tree for each plan. We compared the BUL approach with the CL one for deciding when to consider an execution experiences meaningful for a plan node.

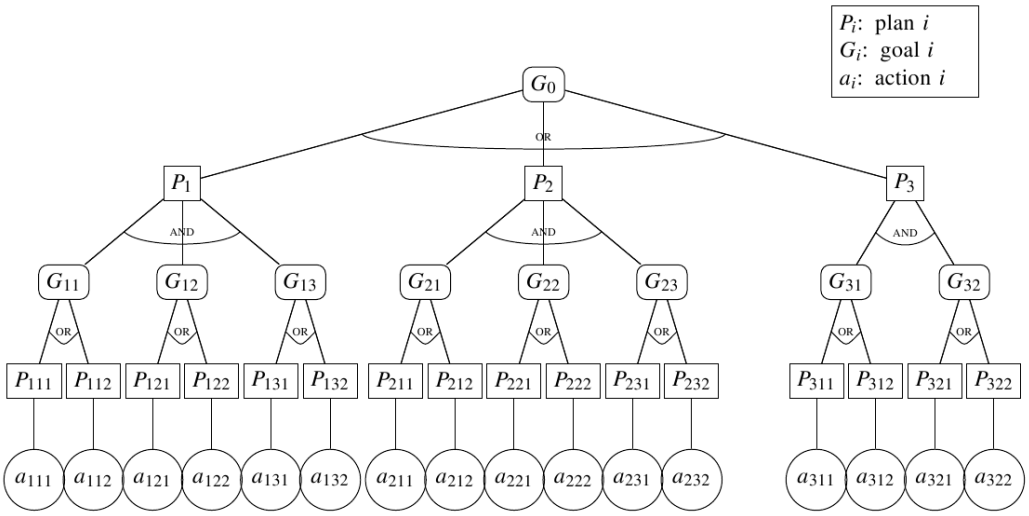

Figure 3: Goal-plan tree used for the experiments

We set up a testbed program with 19 plans, 6 variables, and 4 levels in the goal-plan tree, which is depicted in Figure 3. In the experiments with deterministic environments, each plan leaf node had one action, which succeeded in a particular state and failed in all others. Thus, we expect the characterization of such state to be the context condition to be learnt for the corresponding plan. In non-deterministic experiments, actions also failed 10% of the time in the world states where they are expected to succeed (i.e., where their precondition holds).

To make sure there is indeed space for learning, we ensured that from every possible world state, there is always some successful plan for the top-level goal. For all plans, the context condition initially specified by the programmer was empty (i.e., always true). At each iteration of the experiment, the state of the environment was randomly set and the top goal was posted.

Recall that we have not taken into account the typical BDI failure recovery mechanism. A failure or success of the top level goal is recorded at the end of each iteration. As the agent collects data, decision trees are built at each plan node, using the J48 decision tree algorithm implemented in weka (Witten & Frank, 2005). Currently, we constantly keep rebuilding the decision trees; in future, we expect to use the incremental algorithm of (Swere et al., 2006).

Finally, we recall that, in the experiments reported in this paper, we used the rather simplistic approach of waiting for some number of instances k for determining that a decision tree is considered reliable (see Algorithm 1).

Results

In the deterministic environment, one would expect to learn perfect behavior, and this was indeed corroborated by our experiments. Using BUL and k=35, we the agent achieved 100% success in selecting the right plan. Using CL with k=35, in contrast, the agent never achieved perfect behavior, as top level plans collected false negatives (i.e., a failure that was due to a wrong decision in the tree below, when a success was possible by making a correct decision). Nonetheless, CL managed to achieve perfect behavior when plan selection used a k=50: the additional data was sufficient to counteract the false negatives.

Figure 4 shows the performance level of the agent after learning, for different values of k. Differences are statistically significant for k=10 (p=3.87e–05), though this is no longer the case for $k \geq 20$ (p=0.23, for k=20, p=0.14 for k=35). The figure also indicates the total number of instances required before performance converged, i.e., the decision tree did not change with further training instances. The number of total instances required varies greatly between the CL and BUL approaches. For example, with k=50, CL requires a little over 200 total instances to achieve perfect behavior, whereas BUL with k=35 requires a little over 800 instances. This is to

be expected as, when using BUL, top-level plans in the goal-plan tree must wait for plans below to have collected enough data before collecting their own data.

So, as expected, when using BUL, the agents can learn perfect behavior, but require sufficient number of instances (k>35) before using a tree and a (potentially) large total number of instances. Using CL, on the other hand, an agent can still learn to behave perfectly with higher values of k, but requires a much smaller number of total instances. It is unclear at this stage how the (lack of) complexity (size of the goal-plan tree, number of attributes, complexity of the real context conditions) of the scenario studied affects the relative values of the total instances required.

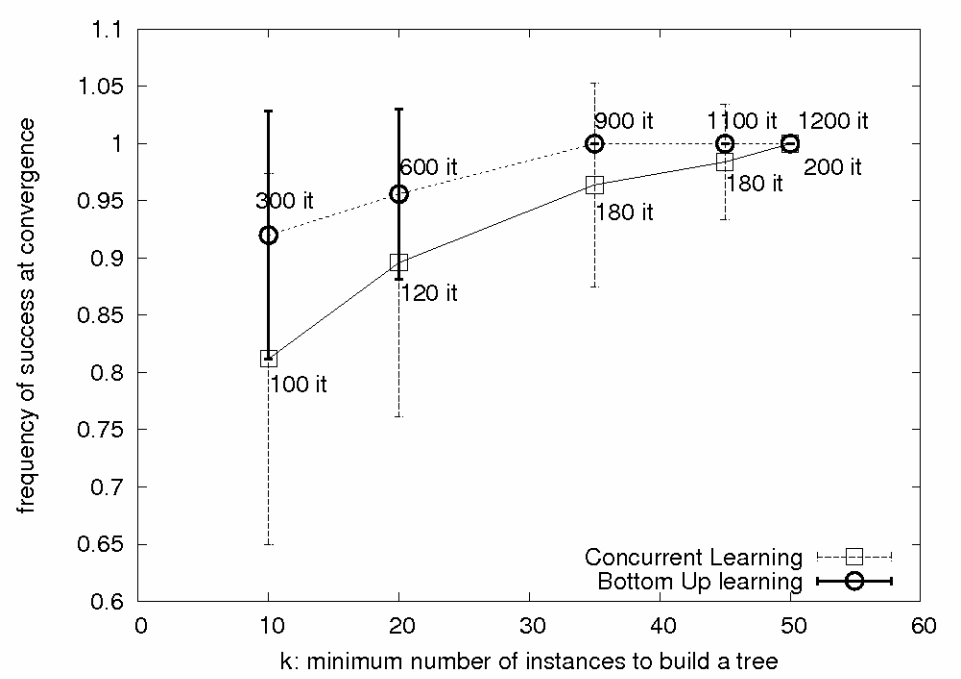

Figure 4: Deterministic environment.

In the non-deterministic environment, we do not expect perfect behavior irrespective of how well agents have learnt the context conditions. Experimentally, with a 10% non-deterministic chance of failure, we can only achieve about 83% success (this can be shown to be correct analytically also). With *k*=50, CL produces an average success rate of 76% and BUL an average success rate of 79.5% (difference not statistically significant). The basic pattern in the non-deterministic case is the same as for the deterministic one. One can start using the decision trees with less data, without degrading performance, if using BUL. However, CL is able to achieve an optimal result faster overall by waiting longer before making any use of the learned information.

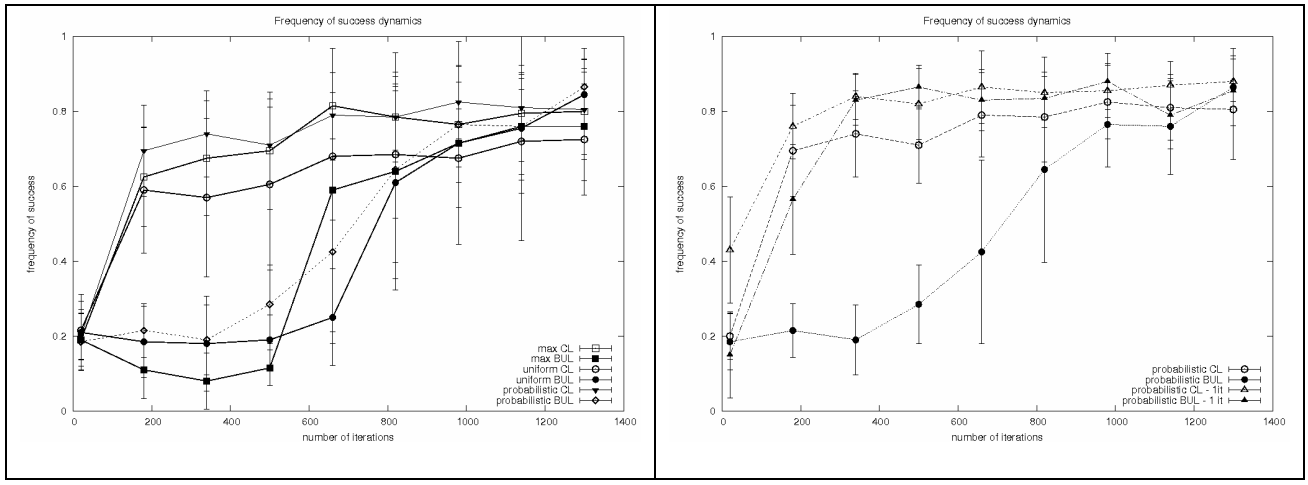

Figure 5: Comparison of the different approached in a non deterministic environment.

The above experiments were conducted using simple selection. Once we introduced probabilistic selection, we observed that CL was able to recover from its previous performance degradation. In the deterministic environment, using CL and *k*=1, probabilistic selection yielded 100% accuracy after a little over 200 instances. Plans that are successful early on are chosen more frequently. In the presence of false negative instances, a plan may be classified as not applicable, but it can still be selected due to the probabilistic selection mechanism. With more

experience, the agent is likely to collect more positive instances, increasing then the likelihood of selecting the plan and therefore "correcting" the corresponding decision tree. For the non-deterministic environment, an average success rate of 83% was achieved after about 350 instances.

The experiments in Figure 5 compare the different selection mechanisms. These graphs are averaged over 10 runs. In the left graph, we set k=20. First, the graph shows that using CL is clearly better early on. The asymptotic behavior of all the mechanisms are identical (at least, no statistical difference). Interestingly, the probabilistic version seems to do somewhat better than the other two selection mechanisms. The right graph compares the use of a different update method when the probabilistic selection is used. The curves noted "1 it" correspond to the case where k=1. One can see that CL in that case is better early on, and as good as BUL later on.

Discussion and Conclusion

In this paper, we explored the use of learning to extend typical BDI agent systems. To implement a BDI agent system, the designer-programmer must provide each plan in the plan library with a context condition, stating when the plan may be used. Writing adequate context conditions for each plan may be a difficult, if not impossible, task. As context conditions play a central role in the overall performance of a BDI system, we propose to analyze past execution experiences to improve the context conditions of the plans. To that end, we assume that the BDI programmer sets initial necessary conditions for the success of each plan; and the task of the learning module is to *refine* such conditions to avoid or minimize re-occurrence of failures. We first studied an idealized setting, where the environment is observable and totally deterministic, in order to get an understanding of the issues and challenges involved in the learning task. We

showed that even within the simplified context, correctly refining the context conditions is not trivial. We then relaxed the assumption on determinism and discussed how to use decision trees to refine the context conditions. In particular, we discussed the issues of data collection in a decision tree and actual use of decision trees for the plan selection process.

Others have also used decision trees as a tool for learning plan context conditions. In (Phung, Winikoff, & Padgham, 2005), inductive learning was used to build a decision tree for deciding which plan to use. The instances observed between the update were used to estimate the accuracy of the tree, and when the tree was successful enough, it was used as the context condition. The problem faced by the learning agent is similar to ours when all initial context conditions are set to true. Their approach, however, is tailored to a particular example, and it is not clear how to generalize it. Hernandez et al. discussed learning from interpretation, which extends learning of decision trees (Guerra-Hernández, Fallah-Seghrouchni, & Soldano, 2004a, 2004b); as ours, their work is preliminary. The approach of (Jiménez, Fernández, & Borrajo, 2008) shares some of our goals, namely to tailor the behavior of an agent to its environment and its particular uncertainty. In the context of planning, the authors propose to use past experiences to learn the conditions of success of individual actions and encode the knowledge learnt in a relational decision tree. When a new problem arises, the planner takes advantage of the new knowledge. Our goal is to provide learning capabilities to a (generic) BDI agent, which generally do not engage in first principles planning. We are interested in learning the context condition of plans, rather than individual actions. When learning only the pre-conditions of actions, which are leaf nodes in a hierarchical structure, one is not faced with the issues and challenges we have discussed in this work.

The learning framework described in this paper has several drawbacks and places for improvement. As noted before, the criterion used to test the accuracy-stability of decision trees is currently simplistic and more sensible criteria need to be tested. We expect more involved criteria would lead to an improvement on the robustness of learning. In addition, we are also interested in using algorithms that are designed to induce decision trees *incrementally* (Swere et al.., 2006, Utgoff et al.., 1997), rather than rebuilding the whole decision tree when a new instance is added to it. We shall also investigate the use of automated HTN planning to enhance the refinement of context conditions by checking for potential successful decompositions. This is because, although initially there may be a full decomposition for a plan *P*, this may not be the case anymore after some context conditions of sub-plans have been refined. One can then perform HTN offline planning to check if there is still a valid decomposition of *P*, and if not, then *P*'s context conditions should also be updated (i.e., further restricted).

Finally, we point out that we have restricted our attention to the learning of plans' context condition. This type of learning has intrinsic limitations due to the locality property of such conditions. In particular, by modifying context conditions, the agent may be unable to learn interactions between plans or new means of achieving its goals. To that end, an account that modifies the meta-level plan selection function or the plan-goal hierarchy itself (e.g., by adding a new plan for a goal) would be necessary.

Acknowledgment

We would like to thanks María Inés Crespo for helping us with the editing of the document and the anonymous referees for their suggestions to improve the paper.

References

Aha, D. W., Kibler, D., & Albert, M. K. (1991). Instance-based learning algorithms. *Machine Learning*, *6*(1), 37-66.

Alonso, E., d'Inverno, M., Kudenko, D., Luck, M., & J.Noble. (2001). Learning in multi-agent systems. *Knowledge Engineering Review*, *16*(3), 277-284.

Booker, L., Goldberg, D., & Holland, J. (1989). Classifier systems and genetic algorithms. *Artificial Intelligence*, *40*, 235-282.

Bordini, R. H., Hübner, J. F., & Wooldridge, M. (2007). *Programming multi-agent systems in AgentSpeak using Jason*. Wiley. (Series in Agent Technology)

Bratman, M. E. (1987). *Intentions, plans, and practical reason*. Cambridge, MA: Harvard University Press.

Bratman, M. E., Israel, D. J., & Pollack, M. E. (1988). Plans and resource-bounded practical reasoning. *Computational Intelligence*, *4*(4), 349–355.

Busetta, P., Rönnquist, R., Hodgson, A., & Lucas, A. (1998). *JACK Intelligent Agents - Components for Intelligent Agents in Java* (Tech. Rep.): Agent Oriented Software Pty. Ltd, Melbourne, Australia. (Available from http://www.agent-software.com)

DeJong, G., & Mooney, R. (1986). Explanation-based learning: An alternative view. *Machine Learning*, *1*, 145-176.

Dennett, D. C. (1987). *The intentional stance*. MIT Press.

Ellman, T. (1989, November). Explanation-based learning: A survey of programs and perspectives. *ACM Computing Surveys*, *21*(2), 163-221.

Georgeff, M. P., & Ingrand, F. F. (1989). Decision making in an embedded reasoning system. In *Proceedings of the international joint conference on Artificial Intelligence* (pp. 972–978).

Guerra-Hernández, A., Fallah-Seghrouchni, A. E., & Soldano, H. (2004a). Distributed learning in intentional BDI multi-agent systems. In *Proceedings of the fifth Mexican international conference in Computer Science (ENC'04)* (pp. 225–232). Washington, DC, USA.

Guerra-Hernández, A., Fallah-Seghrouchni, A. E., & Soldano, H. (2004b). Learning in BDI multi-agent systems. In *Computational logic in multi-agent systems* (Vol. 3259, pp. 218–233). Springer Berlin / Heidelberg.

Hindriks, K. V., Boer, F. S. D., Hoek, W. V. D., & Meyer, JJ. C. (1999). Agent programming in 3APL. *Autonomous Agents and Multi-Agent Systems*, *2*(4), 357–401.

Huber, M. J. (1999, May). JAM: A BDI-theoretic mobile agent architecture. In *Proceedings of the third international conference on autonomous agents, (AGENTS'99)* (p. 236-243).

Jiménez, S., Fernández, F., & Borrajo, D. (2008). The PELA architecture: Integrating planning and learning to improve execution. In *Proceedings of the twenty-third AAAI conference on Artificial Intelligence* (AAAI'08) (pp. 1294–1299). Chicago, USA.

Kaelbling, L., Littman, M. L., & Moore, A. W. (1996). Reinforcement learning: A survey. *Journal of AI Research*, *4*, 237-285.

Kolodner, J. (1993). *Case-based reasoning*. San Mateo, CA: Morgan Kaufmann.

Krause, P. J. (1998). Learning probabilistic networks. *The Knowledge Engineering Review*, *13*(4), 321-351.

Mitchell, T. M. (1997). *Machine learning*. McGraw Hill.

Panait, L., & Luke, S. (2005). Cooperative multi-agent learning: The state of the art. *Autonomous Agents and Multi-Agent Systems*, *11*(3), 387–434.

Phung, T., Winikoff, M., & Padgham, L. (2005). Learning within the BDI framework: An empirical analysis. In *Proceedings of the 9th international conference on knowledge-*

based intelligent information and engineering systems (KES'05),LNAI volume 3683 (p. 282). Melbourne, Australia: SpringerVerlag.

Quinlan, R. J. (1986). Induction of decision trees. *Machine Learning, 1*, 81-106.

Rao, A. S., & Georgeff, M. P. (1992). An abstract architecture for rational agents. In *Proceedings of the third international conference on principles of knowledge representation and reasoning* (KR'92) (p. 439-449). San Mateo, CA, USA.

Rumelhart, D., Hinton, G., & Williams, R. (1986). Learning internal representations by error propagation. In D. Rumelhart & J. McClelland (Eds.), *Parallel distributed processing* (Vol. 1). Cambridge, MA: MIT Press.

Sardina, S., de Silva, L. P., & Padgham, L. (2006). Hierarchical planning in BDI agent programming languages: A formal approach. In *Proceedings of autonomous agents and multi-agent systems (AAMAS'06)* (pp. 1001–1008). Hakodate, Japan.

Swere, E., Mulvaney, D., & Sillitoe, I. (2006). A fast memory-efficient incremental decision tree algorithm in its application to mobile robot navigation. In *Proceedings of the 2006 IEEE/RSJ international conference on intelligent robots and systems* (pp. 645–650).

Tuyls, K., & Nowé, A. (2006). Evolutionary game theory and multi-agent reinforcement learning. *The Knowledge Engineering Review*, *20*(1), 63-90.

Utgoff, P. E., Berkman, N. C., & Clouse, J. A. (1997). Decision tree induction based on efficient tree restructuring. *Machine Learning*, *29*(1), 5–44.

Witten, I. H., & Frank, E. (2005). *Data mining: Practical machine learning tools and techniques* (2nd Edition ed.). Morgan Kaufmann, San Francisco.