Citation:
Phung, T, Winikoff, M and Padgham, L 2005, 'Learning within the BDI framework: an empirical analysis', in Knowledge-Based Intelligent Information and Engineering Systems, Part 3, Melbourne, 14-16 September 2005.

Learning within the BDI Framework: An Empirical Analysis

Toan Phung, Michael Winikoff, and Lin Padgham

{tphung,winikoff,linpa}@cs.rmit.edu.au
RMIT University, School of Computer Science and IT, Melbourne, Australia

Abstract. One of the limitations of the BDI (Belief-Desire-Intention) model is the lack of any explicit mechanisms within the architecture to be able to learn. In particular, BDI agents do not possess the ability to adapt based on past experience. This is important in dynamic environments since they can change, causing methods for achieving goals that worked well previously to become inefficient or ineffective. We present a model in which learning can be utilised by a BDI agent and verify this model experimentally using two learning algorithms.

1 Introduction

Intelligent Agents are a new way to develop software applications. They are an amalgam of Artificial Intelligence (AI) and Software Engineering concepts that are highly suited to domains that are inherently complex and dynamic [1, 2]. Agents are autonomous in that they are able to make their own decisions. They are situated in an environment and are reactive to this environment yet are also capable of proactive behaviour where they actively pursue goals. BDI (Belief Desire Intention) agents are one popular type of agents which support complex behaviour in dynamic environments [3, 4].

BDI agents use plans to achieve goals, based on the current environment. When a BDI agent encounters a problem where it can not complete the current plan, it will stop executing that plan, re-assess the situation based on the updated environment and select a new plan from a plan library. This provides a level of adaptivity to the changing world. However it does not provide any adaptation based on past experience. Such an ability can be important in dynamic environments which may change in ways not foreseen by the developer, causing methods for achieving goals that worked well previously to become inefficient or ineffective. Our work aims to improve BDI agents by introducing a framework that allows BDI agents to alter their behaviour based on past experience, i.e. to learn.

We have chosen a fire fighting scenario as our experimental domain. This system simulates a city that has been affected by fire and will allow us to learn answers to questions such as *"Based on past experience, what's the best fire extinguisher to use now?"* We do not develop new learning techniques, rather, the contribution of this paper is to propose a model for integrating learning into BDI agents, and to experimentally validate that this model allows agents to improve their performance over time.

2 Background

2.1 The BDI Agent Architecture & JACK

The Belief-Desire-Intention (BDI) [3] model is based on philosophical work by Bratman, which stresses the importance of *intentions*, defining the (human) agent's current approach, as critical in intelligent behaviour, as well as *beliefs* and *desires*. The computational model of agency developed by Rao and Georgeff [4] based on Bratman's work focusses on (software) agents which are situated in an environment, receiving stimulus in the form of *events* and acting based on *plans* in a plan library.

JACK[1] is a Java-based intelligent agent toolkit used to implement BDI agents. There are four main components to a JACK system: agents, events, plans and beliefsets. When a JACK agent receives an event, which may correspond to a goal, it will refer to its *plan library*. Plan libraries act as a repository of plans. Plans consist of (i) a *trigger* which indicates which event they are relevant to; (ii) a *context condition* which describes the situation in which they are applicable; and (iii) a *plan body*. The plan body may contain both *sub goals* and actions. There may be multiple plans associated with any given goal or event. If a plan fails during execution, the agent checks to see whether other plans are applicable. Beliefsets can be viewed as relational databases, i.e. sets of tuples.

2.2 Inductive Logic Programming and Alkemy

Inductive logic programming (ILP) is a means of computationally achieving induction [5]. Induction can be defined as: given a set of positive examples, a set of negative examples, some background knowledge and a hypothesis language, find a predicate definition represented in the hypothesis language such that all positive examples and none of the negative examples are described.

Alkemy [6] is a symbolic inductive learner written in C++. It uses Inductive Logic Programming to produce a decision tree (see figure 1). Each node in the decision tree generated by Alkemy contains a higher order function that takes an *Individual* and returns a Boolean. For example, consider the root node in figure 1. The composition `projIntensity . eqHigh` is the function that takes an Individual and returns true iff its Intensity is High. The whole root node expression denotes a function that takes an Individual and returns true iff its Intensity is High and its Weather is Hot.

3 Learning in the BDI Framework

Our framework consists of four major components: the JACK system, the *Learning Formatter*, the *Learning Component* and the *Knowledge Extractor*. Figure 2 shows our conceptual model.

The flow of information begins with the *JACK agent*. This agent stores its experiences in the *History* beliefset. When enough history accumulates, *Learning Formatter* converts the History and *Background Knowledge* (provided by the agent designer) into an input suitable for the *Learning Component*. Learnt data is returned and converted into

[1] http://www.agent-oriented.com

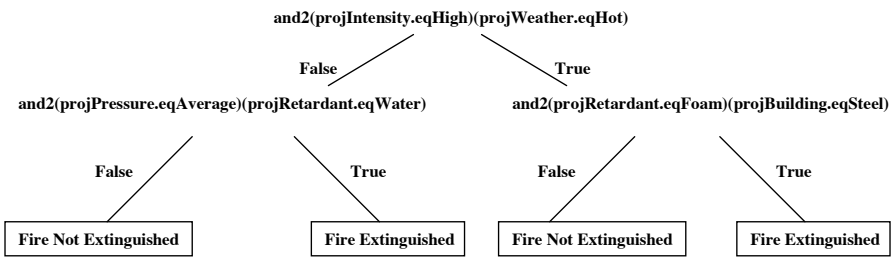

Fig. 1. Higher Order Function Tree

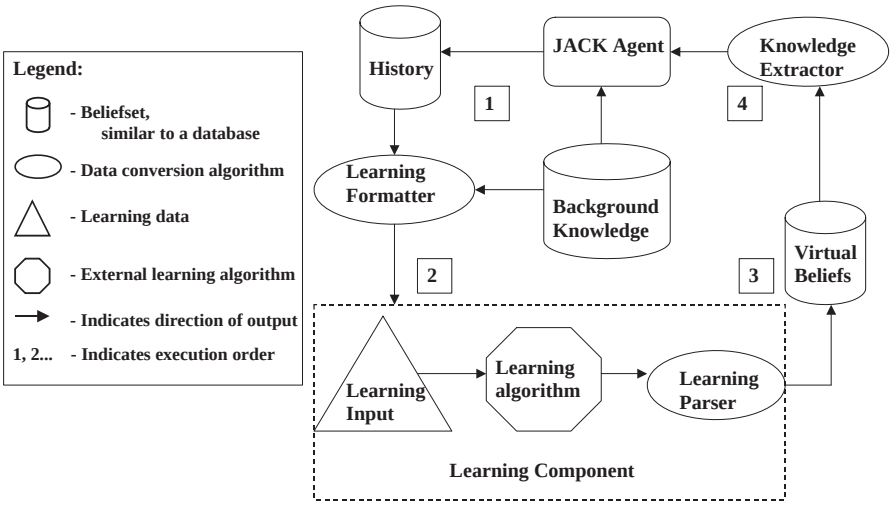

Fig. 2. BDI Learning Model

Virtual Beliefs by the *Learning Parser* which translates Alkemy's textual output into a binary tree. These virtual beliefs are queried by the *Knowledge Extractor*, allowing the agent to reason historically.

The History stored by the agent is a set of tuples containing the *state of the fire*, the *outcome* and a *retardant*. For example, the History tuple ⟨windy, concrete, high, success, water⟩, represents that it was a windy day when water was successfully used to extinguish a concrete building burning with a high intensity. Actually, in order to experiment with different search space sizes the *state of the fire* varies from 3 to 11 elements.

The operation of the *Knowledge Extractor* involves the following steps: (1) Estimate the accuracy of the tree produced by Alkemy; (2) If the accuracy is "good enough" (see below) then use the recommendation produced by the tree, else explore.

Estimating the accuracy of the decision tree is done by checking the tree's predictions against the actual outcome for all of the recent fires that the agent has fought which

have not yet been used for learning. This gives a number between 0 and 1. For example, if there are 37 recent fires that have not yet been learned from, and for 32 of them the decision tree correctly predicts the outcome, then the estimated accuracy of the Alkemy tree is $32 \div 37 \approx 0.865$.

Producing a recommendation from the Alkemy decision tree is done as follows. First, the Knowledge Extractor scans the higher order function tree to see what values exist for the *retardant* variable. If none are found, then the agent has had no relevant prior experience and will return *unknown* or a default value. If values are found, then the Knowledge Extractor will record every unique value[2]. This forms a set *potential* retardants to use. For each potential retardant the Knowledge Extractor uses the decision tree to predict the outcome of using that retardant on the current fire. Those retardants for which the decision tree predicts a successful outcome are retained as the tree's recommendation.

Determining whether the tree's accuracy is "good enough" is done in a number of ways: using a static threshold (e.g. 0.5), using a dynamic threshold with analogous reasoning, or using a dynamic threshold without analogous reasoning. When using a dynamic threshold, the threshold is adjusted up or down by considering the subset of the fires previously encountered which are either the same (the "without analogous reasoning" case) or "similar" (the "with analogous reasoning" case). Adjusting the threshold is done as follows: for each fire that is considered we adjust the threshold up if the fire was successfully fought, and down if it was not successfully fought. The formula used to calculate dynamic thresholding is:

$$\text{threshold} = \text{static threshold} - \frac{\text{successful cases} - \text{failed cases}}{2 \times \text{total cases}}$$

If analogous reasoning (also termed "*Simile*") is used then a previously encountered fire is considered to be "similar" if it was successfully fought and is harder than the current fire (because fighting the current, easier, fire can be assumed to succeed) or if it was unsuccessfully fought and is easier than the current fire. For example, suppose the agent is fighting a fire in *Hot* weather where the building is made of *Wood* and the fire is burning with a *High* intensity. A previously fought fire that was on a *Mild* day, involved a *Steel* building, and was a *Medium* intensity fire is easier than the current fire. If a particular retardant was unsuccessfully used on the previous, easier, fire than the simile algorithm will reason that the retardant in question is probably a bad choice for the current, harder, fire.

Exploration is implemented by subtracting all the previously seen retardants from the full list of known retardants, and selecting a random retardant from the result. If the result is empty then a random previously seen retardant that is not recommended by the tree is chosen.

In addition to using Alkemy, we also experiment with a simpler learning mechanism that simply computes for each retardant its effectiveness:

$$\text{effectiveness} = \frac{\text{successes} - \text{failures}}{\text{total}}$$

[2] As well as an additional "none-of-the-above" value.

The retardant with the highest effectiveness is then selected. There are several variants of this depending on whether one considers all past fires, or only past fires similar to the current fire. Note that this simpler mechanism bypasses the learning component depicted in figure 2, since it only requires the agent's history.

4 Experiments

Experiments were conducted within the fire fighting domain to answer the following questions: (1) how effective are various learning mechanisms on BDI agents? (2) what effects do dynamic thresholding and Simile have on learning? and (3) how is the performance of the agent affected by the size of the search space?

Each experiment involved 40 runs, where a run involved a learning agent fighting 1000 fires using one of five retardant types. The fire fighting agent is given no initial past experiences. Fire states were randomly generated using a random number generator that was initialised with a different seed for each run. Alkemy is invoked every 50 fires. The performance of the agent is measured by the percentage of fires extinguished over a given set of fires.

To determine whether a fire is successfully extinguished, we convert every symbolic fire state into a numeric representation and compare that value to a set of rules. In order to do this, every variable is given its own 'difficulty' score. This represents how 'hard' a particular tuple variable is, thus the difficulty score for a entire fire is the sum of the difficulty scores in the fire tuple. Symbolic-to-numeric conversion is done to allow us to easily vary the complexity of the domain. The complexity of the search space is varied from an initial 1440 possible fire states to 2,304,000 by increasing the number of variables in the fire state from 3 through to 11.

All graphs were plotted with the mean of the 40 experimental runs. Each point on the '% of Fires Extinguished' graph represents the success rate over the most recent 50 fires.

4.1 Discussion

Clearly, learning is beneficial to the agent's performance. For the smaller search space (576,000) the statistical method without Simile does best, followed by the Statistical method with Simile, then Alkemy. For the larger search space (2,304,000) Alkemy outperforms all other learning methods by 6%-10% followed by the Statistical method (with Simile not making much of a difference). Overall, learning in the smaller search space yields a 10% improvement over no learning with a 27% improvement in the larger search space.

Although not represented on the graphs, using dynamic thresholding and Simile did not make a difference to Alkemy's success rate. This is because the decision tree rapidly becomes quite accurate, resulting in the slight adjustments to the threshold made by dynamic thresholding not being significant.

Comparing the different search space sizes, as expected, the statistical method's performance degrades as the search space size increases. However, Alkemy's performance doesn't appear to be significantly affected by the search space, and in fact Alkemy does

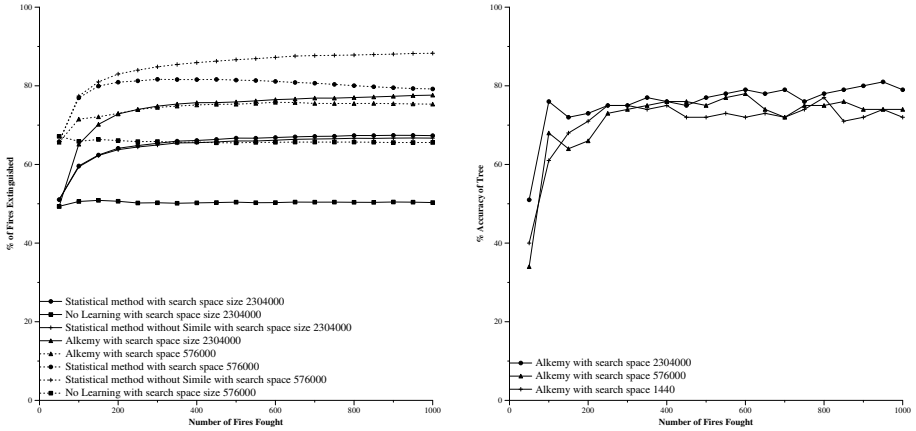

Fig. 3. Experimental Results

slightly better in terms of % of fires extinguished when the search space is larger. We intend to investigate this counter-intuitive result further.

With regard to the second graph (right side of figure 3), the accuracy of the Alkemy tree as measured by the agent is quite erratic and never rises above 81%. This may be because we convert symbolic states into numeric values and the fact that Alkemy is a symbolic learner. We intend to explore this further.

Although Alkemy extinguishes more fires than Statistical learning by an average of 10% in the more complex domain, this comes at a time cost 4 times greater than that of Statistical learning. The Statistical method out-performs Alkemy in the simpler domain, highlighting the fact that complex and powerful learners such as Alkemy are not always necessary.

For both search spaces (2,304,000 and 576,000) Alkemy took on average a little over a minute (61-65 seconds) to induce a decision tree from 500 fires.

5 Related Work

Similar systems to what we propose include SOAR [7], a rule based agent system that uses *chunking* to create plans. Chunking is executed whenever *impasses* occur. An impass is when an agent cannot solve a problem. Our model is different in that we learn new information regardless of problems occurring, which allows for exploratory learning.

The Case-Based BDI system in [8] is similar to our model where it considers past cases. They use a *concept hierarchy* to find information on the WWW if no similar cases are found while we assume no additional information sources and hence use *Simile* to reason further on existing information. The notion of ‘easier‘ and ‘harder‘ for case similarity is absent in [8] however their model applies case reasoning on agent beliefs while we do not.

The system proposed by [9] uses a combination of explanation based learning (EBL) and ILP. EBL uses only one past case to generalise a rule while our statistical method considers all past cases. Another difference is our model uses Simile to filter cases before an ILP system is called.

Prodigy [10] is a planning and learning system that implements many learning algorithms including case-based reasoning and induction. However, their work is not based on the BDI framework.

6 Conclusions & Future Work

We have presented a model that introduces learning into the BDI framework. This model allows beliefs to be generalised through inductive learning and statistical tallying. We have developed and experimentally tested, two analogous reasoning algorithms which use contextual and relative reasoning to alter agent behaviour according to past experience.

Currently, the issue of *when* to learn is addressed by means of a numeric threshold on the number of fires fought. This static technique may greatly over/under utilise a potentially expensive[3] learning process and may be improved by considering the frequency of past successes/failures. We also plan to develop a more effective Simile matching scheme.

Acknowledgements: We would like to thank the Smart Internet CRC for their support with this project.

References

1. N. R. Jennings: An Agent-based Approach for Building Complex Software Systems. Communications of the ACM **44(4)** (2001) 35–41
2. Wooldridge, M.: An Introduction To MultiAgent Systems. first edn. John Wiley and Sons Ltd (2002)
3. Bratman, M.E.: Intentions, Plans, and Practical Reason. first edn. Harvard University Press, Cambridge, MA (1987)
4. A. S. Rao, M. P. Georgeff: BDI-Agents: From Theory to Practice. In: Proceedings of the First International Conference on Multiagent Systems. (1995)
5. Muggleton, S.: Inductive Logic Programming. first edn. Academic Press (1992)
6. Kee Siong Ng: Alkemy: A Learning System Based on an Expressive Knowledge Representation. Available from http://users.rsise.anu.edu.au/~kee/Alkemy/ (2004)
7. J.E. Laird, A. Newell, P.S. Rosenbloom: SOAR: An Architecture for General Intelligence. Artificial Intelligence **3** (1987) 1–64
8. C. Olivia, C.F Chang, C.F Enguix, A.K. Ghose: Case-Based BDI Agents: An Effective Approach for Intelligent Search on the World Wide Web. In: AAAI Spring Symposium. (1999)
9. E. Alonso, D. Kudenko: Logic-Based Multi-Agent Systems for Conflict Simulations. In: Proceedings of the 5th UK Workshop on Multi-Agent Systems UKMAS'00. (2000)
10. M. Veloso, J. Carbonell, A. Perez, D. Borrajo, E. Fink, J. Blythe: Integrating Planning and Learning: The PRODIGY Architecture. Journal of Experimental and Theoretical Artificial Intelligence **7(1)** (1995)

[3] In terms of computational resources