Heuristic Rule Induction for Decision Making in Near-Deterministic Domains

Stavros Korokithakis and Michail G. Lagoudakis

Intelligent Systems Laboratory
Department of Electronic and Computer Engineering
Technical University of Crete
Chania 73100, Crete, Greece
stavros.korokithakis.09@ucl.ac.uk, lagoudakis@intelligence.tuc.gr

Abstract. A large corpus of work in artificial intelligence focuses on planning and learning in arbitrarily stochastic domains. However, these methods require significant computational resources (large transition models, huge amounts of samples) and the resulting representations can hardly be broken into easily understood parts, even for deterministic or near-deterministic domains. This paper focuses on a rule induction method for (near-)deterministic domains, so that an unknown world can be described by a set of short rules with well-defined preconditions and effects given a brief interaction with the environment. The extracted rules can then be used by the agent for decision making. We have selected a multiplayer online game based on the SMAUG MUD server as a model of a near-deterministic domain and used our approach to infer rules about the world, generalising from a few examples. The agent starts with zero knowledge about the world and tries to explain it by generating hypotheses, refining them as they are refuted. The end result is a set of a few meaningful rules that accurately describe the world. A simple planner using these rules was able to perform near optimally in a fight scenario.

1 Introduction

Intelligent agents that learn to make decisions in unknown environments using reinforcement learning [1] focus on the diversity of action effects in different states. The agent explores the world, observing the actions it took at each state and their immediate effects, and eventually learns the action choices that yield the desired long-term outcome. This means that the agent would have to observe more or less every state and perform every action in each state in order to see how they contribute to the desired outcome. This approach quickly becomes intractable in the real world, which nevertheless in many cases exhibits some regularity. Induction learning, on the other hand, aims at deriving generic rules from a few experiences, which can then be applied to previously unseen situations to produce meaningful predictions of the outcomes that will be observed [2].

Multiplayer online games (Multi-User Dungeons or MUDs) represent a large class of moderately complex abstractions of the real world. Input in these games

S. Konstantopoulos et al. (Eds.): SETN 2010, LNAI 6040, pp. 339–344, 2010.

takes the form of commands, which a player enters to perform actions in the world. Each command effects a change in the state of the world which is reflected on the observed variables, and this change is unlikely to be arbitrary. For example, a `fight` variable (boolean) may indicate whether the player is currently fighting an enemy and a `health` variable (integer) may reflect the player's current health. The player can influence these variables by performing commands, namely `strike <opponent>` may cause the player to engage in a fight (`fight` becomes `True`) and the enemy to take some amount of damage. A player currently in a fight can expect to take an amount of damage himself from an enemy fighting back, regardless of his actions. The fighting player's `health` will diminish and he may die if it reaches 0, however he may increase his health by the `heal` command, which is subject to constraints in itself, and so on ...

Our goal is to design agents which are able to discover enough aspects of the world to achieve a list of desired outcomes. We must, therefore, find a way for the agent to understand the effect that his actions have on the world and on its current state. For example, our agent must learn that `heal`ing will increase its health and `fight`ing will decrease it, but will eventually provide him with gold, if he wins. We also want the agent to be able to generalise the rules he discovers and then refine the circumstances he believes they apply to, so that he goes from the general to the specific, instead of the other way around.

2 Decision Making in Near-Deterministic Domains

The world is typically described by a collection of state variables $s_1, s_2, \ldots, s_n$; $S(t) = \big(s_1(t), s_2(t), \ldots, s_n(t)\big)$ reflects the state of the world at time t and $a(t)$ the action taken by the agent at time t. The goal of our agent is to understand how each action impacts the state. We assume an underlying (unknown) transition model $T : S(t+1) = T\big(S(t), a(t)\big)$, which is both Markovian (state transitions depend only on the current state and action and not on the past history) and deterministic (there is only one outcome for a given state and action pair). We must, therefore, discover **what** changes each action imparts on the state in each case, and also **when** these occur, i.e. the conditions under which they occur.

Our goal is to infer a transition model for the world that takes the form of a collection of rules. The general form of these rules is the following:

> *If action a is taken at time t, then a subset of state variables $\{s_i, s_j, s_k, \ldots\}$ will change by $\{d_i, d_j, d_k, \ldots\}$ at time $t+1$, provided that the values of another subset of state variables $\{s_l, s_m, s_n, \ldots\}$ are (not) $\{v_l, v_m, v_n, \ldots\}$.*

Our approach is predicated on the assumption that the world is mostly deterministic (i.e. the same action will not bring radically different results, if performed at two different times in the same state), but we must account for noise, as our measurements might be noisy. Another assumption we make is that there are weak dependencies, i.e. an action affects only a small number of state variables,

and that the preconditions under which this happens involve only a small number of state variables as well. Both assumptions are valid for MUDs in general.

Related Work. Salzberg [3] developed HANDICAPPER, a system for predicting horse race outcomes using heuristic inductive learning methods, which fares markedly better than human experts and chance. He describes various heuristics at a high level, providing inspiration for our work, however it was not clear how to combine them at an algorithmic level and apply them to our domain. Amir and Chang [4] developed an exact solution for identifying actions' effects in partially observable STRIPS domains. Their methods apply in other deterministic domains with conditional effects, but may be inexact, as they produce false positives, or inefficient, as the resulting model can grow arbitrarily).

The MUD Domain. Our test domain is a subset of the full MUD domain that focuses on combat situations. We have selected the following state variables:

Health is an integer in the range of approximately $[0 - 1000]$. It reflects the player's well-being and it is generally desired that its value is as high as possible. If a player's `health` falls to zero, the player dies.

Mana is an integer in the range of approximately $[0 - 700]$ and reflects the amount of available "magic power" the player has for casting spells.

Fight is a Boolean variable indicating whether the player is in combat or not. When in combat, the player will usually take (and deal) damage.

Enemy health is an integer variable in the range of [0-11], which indicates the current health of the enemy. If it reaches zero, the enemy dies.

There are four combat-related actions available to the agent:

Pause is a control action that does nothing. We include it so that we can see what happens in the world without our intervention.

Strike opponent is an aggressive action that causes damage to the opponent. Additionally, the angry opponent starts a fight and `fight` changes to `True`.

Heal causes our `health` and `mana` to increase by certain amounts. Obviously, if these variables have already reached their maxima, this action has no effect.

Cast spell is another aggressive action that causes damage to the opponent. It also turns `fight` to `True` and makes `mana` to diminish by some amount.

3 Heuristic Rule Induction

In this section, we propose a heuristic algorithm for learning rules that attempt to capture the preconditions and effects of actions using a series of observed interactions with the world. An agent can then use these rules to plan his actions.

Rule Induction. The first step is to construct rules for each action. We want to infer the changes that an action brings about to the world, so it makes sense to start by grouping all the results by actions. Since there may be several outcomes for an action, we allow for multiple disjoint pairs of preconditions and effects

```
update_action_rules (S(t − 1), a(t − 1), s(t))
    //  S(t − 1): previous state, a(t − 1): action performed, S(t): current state
    δ(t) ← S(t) − S(t − 1)
    if (δ(t) ∈ δ_{a(t−1)})              // If the changes in the state are in the deltas set,
        reinforce(a(t − 1), δ_t)        // reinforce the already existing delta.
    else
        δ_{a(t−1)} ← δ_{a(t−1)} ∪ δ(t)  // Otherwise, add it to the deltas set.
    for each δ ∈ δ_{a(t−1)}
        if δ = δ(t)
            pos_precon(a(t − 1), δ, S(t − 1)) // Positive precondition.
        else
            neg_precon(a(t − 1), δ, S(t − 1)) // Negative precondition.
```

for each action. Initially, the outcomes for each action are empty and they are updated as the data is processed. The update step of the algorithm takes as input a state pair and an action. A state pair consists of a state, $S(t)$, and the one immediately preceding it, $S(t-1)$. The action $a(t-1)$ is the action that the agent took between those two states. The algorithm then notes the differences between the two states in a vector δ and checks if this particular outcome has been seen before. If so, trust in it is reinforced by increasing a confidence value. If not, it is added to the set of outcomes for the current action. The next step is to infer a priori necessities, which we call *preconditions*. Briefly, the current state is added as a positive precondition to the current outcome and as a negative precondition to all the others. More specifically, if an action $a(t-1)$ led to a certain outcome $\delta(t)$, while a state variable $s_i(t-1)$ had the value v, but never when it had any other value, then we can infer that, for the action $a(t-1)$ to produce the outcome $\delta(t)$, the variable s_i must have the value v. The opposite is true for negative preconditions, i.e. ones that require that the state value **not** be v. State variables that do not change between $S(t-1)$ and $S(t)$ are discarded. The rule induction algorithm, shown in the box above, creates the deltas for each action and a measure of confidence for each rule.

Merging. To counter duplicate outcomes that might be produced due to noise, we take a merging step. This step merges any two outcomes of a rule that share the same preconditions. For example, an action might sometimes cause a variable to increase by 10 and sometimes by 5. In this case, the two rules could be merged by taking the average 7.5 as their numeric outcome. It stands to reason that, if a result δ_i of an action a_i has priors p_i and another result δ_j of the same action has the same priors, then the two results must be different instances of the same mechanism.

Planning. Initially, since the player will know nothing about the actions, it can fall back on a purely random player, which will, nevertheless, aid in exploration. The agent can then decide to leverage exploitation vs exploration as it generates new rules. One way to plan would be deciding which target state we would like to be in in the long term and using the rules as transformations on the current state to reach our target. This process is known as *planning* and the action chain is called a *plan*.

4 Experimental Results

The following results were obtained by first running a random player for 600 actions to collect data and then running our algorithm off-line to learn rules. The learned rules along with the actual rules are shown below.

The "pause" action is not a real action, but merely a way of telling what happens if we do nothing. When fighting, the player takes some damage, but nothing happens when not fighting. In this case the preconditions are wrong; this action does not require `mana` to have any specific value. The erroneous preconditions caused the merging step to err, creating two outcomes instead of one. Note that our agent has not seen any data of "pause" when not in a fight.

```
Actual : "pause" decreases health by about 40 units when fighting, derived:
Confidence 18: Influences: health by -48.61, mana MUST NOT be 1
Confidence 14: Influences: health by -35.00, mana MUST NOT be 2
```

The "cast spell" action is a way for the player to cast an offensive spell on the target, draining its `health`. We can see that the effects are once more derived correctly, but not the priors. This is the result of seeing too few useful states, but this problem can be addressed with directed exploration.

```
Actual : "cast spell" causes mana to decrease by 8 and the opponent to take damage, derived:
Confidence 142: Influences: mana by -8.00, health by -64.57, enemy_health by -1.00
Confidence 131: Influences: mana by -7.72, enemy_health by -1.0, health MUST NOT be 2
```

The "heal" action increases the `health` by 200 and `mana` by 60; `health` is and `mana`, but we see that the detected `health` increase is 116 on average, because the enemy deals damage which decreases our `health` at every turn.

```
Actual : "heal" health to increase by 200 and mana to increase by 60, derived:
Confidence 618: Influences: mana by 58.0, health by 116.96, enemy_health MUST NOT be 0
Confidence 158: Influences: mana by 16.0, health by 116.53, health MUST NOT be 2
```

The "strike" action deals damage to the opponent and starts a fight if there isn't one. The rule (not shown due to space limitations) is correctly derived and the change of the state in `fighting` is correctly interpreted. The decrease in `health` is, again, because we get dealt damage by the opponent when in a fight.

To obtain decision making results, we used a rudimentary planner that acts in very basic ways to ensure its survival, yet makes full use of the rules we have created. It was not within the scope of this paper to utilise a competent planner. At each planning step, the current state is inspected and the actions whose preconditions in the learned rules are not satisfied are culled. Afterwards, the planner decides which action to use according to its directives and the rules. Figure 1 (a) shows the health fluctuations of the random player. Health fluctuates wildly, as we would expect. Figure 1 (b) shows the performance of a player that plans using the rules learned off-line. The health of this player fluctuates minimally, between about 880 and 1000. Figure 1 (c) shows the performance of a player that starts out with no knowledge (essentially a random player), but periodically uses its samples to learn rules. We can see that after only 150 samples it has learnt rules good enough to play near-optimally, and its health never drops below 900 (except for cases of large damage).

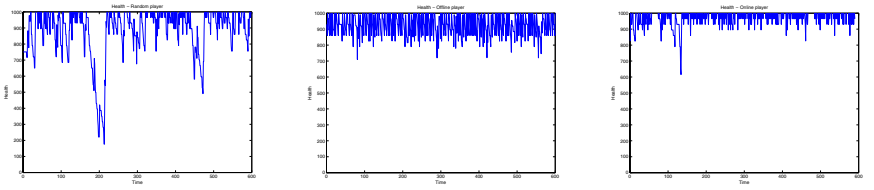

Fig. 1. Health fluctuations of a (a) random, (b) learned, and (c) online player

5 Discussion and Conclusion

The proposed algorithm operates at a high level, which means that the rules it generates are very compact. It not only infers which variables changed, but also *how* they changed, usually with specific values. This offers an obvious advantage to planning, because the planner can predict with greater accuracy how an action is going to alter the current state. In addition, the algorithm starts generating rules from as early as the first datum. These rules may later be revised, superseded, or reinforced, but there is virtually no bootstrapping period as we can begin to perform actions that are more relevant to the environment almost immediately. Finally, it provides the planner with confidence measurements on each rule. This means that the planner can *actively* decide if it wants to experiment with another area of the search space.

As usual, there are always tradeoffs, and the proposed algorithm has its own limitations. It is not well suited to purely stochastic environments. If the world is purely stochastic, the advantage of precise reporting our algorithm offers will be lost. Furthermore, the algorithm cannot handle multiple variable dependencies. If a result is dependent on the specific *combined* value of two or more variables, our algorithm will not detect this correctly.

Inspired by the way humans approach problems, we have presented a method to infer rules about the world and generalise examples of data so that we can apply the actions to unknown circumstances. We have demonstrated the advantages of our algorithm, namely the fact that it is online, is amenable to directed learning, and fast with low memory requirements. The results have been demonstrated, and they show a marked increase of the ability of the player to function in the game, as compared to the random player.

References

1. Kaelbling, L.P., Littman, M.L., Moore, A.W.: Reinforcement learning: A survey. Journal of Artificial Intelligence Research 4, 237–285 (1996)
2. Wexler, M.: Embodied induction: Learning external representations. In: AAAI Fall Symposium, pp. 134–138. AAAI Press, Menlo Park (1996)
3. Salzberg, S.: Heuristics for inductive learning. In: Proceedings of the International Joint Conference on Artificial Intelligence, pp. 603–609 (1985)
4. Amir, E., Chang, A.: Learning partially observable deterministic action models. Journal of Artificial Intelligence Research 33, 349–402 (2008)