ARO: A Memory-Based Approach to Environments with Perceptual Aliasing

Ryan Regier, Owen Price, Alex Hadi, Zachary Faltersack and Andrew Nuxoll
University of Portland
5000 North Willamette Boulevard
Portland, Oregon 97203
engineering@up.edu

Abstract

In order to integrate machine learning and knowledge engineering, it is necessary to store both learned and engineered knowledge in a common format that can be used by an agent to make intelligent decisions. Such integration is much more useful in data sparse environments, where traditional machine learning techniques fail. To this end, we define a rule structure that stores patterns of cause and effect. These rules can easily embed expert knowledge on causal relationships. We will then present ARO, an algorithm for learning and using rules to navigate data sparse environments. Our results show that ARO is more effective than other known solutions to this problem. ARO has the additional advantages that it more easily integrates with knowledge engineering techniques and uses no hyperparameters.

Background

In environments with an abundance of data, machine learning agents are effective at learning patterns, even those not known by experts in the domain. However, in data-sparse environments, traditional machine learning agents fail (Chrisman 1992). We believe that this kind of environment could be ideal for the integration of knowledge engineering with machine learning techniques. One such environment is the Blind Finite State Machine environment. In this environment, the agent navigates a finite state machine (FSM) (Hopcroft, Motwani, and Ullman 2006) to reach a goal state. The machine is designed so that the agent can reach the goal from any state (no dead ends) but otherwise the transition table is randomly generated. Upon reaching the goal the agent is immediately moved to a randomly selected, non-goal state and informed of its success. Thus the agent knows when it reaches the goal but can never actually take an action in the goal state.

The task seems trivial at first. However, it is greatly complicated because the agent is only aware of the following:

- the FSM alphabet (available actions)
- a goal sensor indicating when it reaches the goal

Specifically, the agent is not aware of the following:

- the number of states
- what state it currently is in
- the transition table of the FSM

The agent repeatedly performs the task in a given FSM and its success is measured by how many actions it takes to reach the goal at each iteration.

In such an environment, a traditional machine learning agent, e.g., Q-Learning (Sutton, Barto, and others 1998), is unable to learn effective behavior. Such an agent attempts to learn the action that yields the maximum expected utility for each possible sensory input. However, in this environment, all states (except the goal state) appear identical to the agent. Furthermore, there is no reward function to indicate to the agent that it has performed particularly poor actions such as looping. So, the agent can only learn a single "best" action to take in any non-goal state.

The situation changes once the agent is given additional sensors beyond the goal sensor. Consider an environment like Blind FSM but the agent also has a binary sensor that tells it when it is currently in an odd-numbered state or an even-numbered state (presuming the states were arbitrarily enumerated). We shall henceforth refer to this as the OSFSM (Odd-Sensor Finite State Machine) environment. Traditional machine learning algorithms still perform poorly in OSFSM since they can only distinguish three categories of states: odd-numbered, even-numbered or the goal. However, as more and more sensors are provided a traditional machine learning algorithm becomes more and more effective. Thus, there is a continuum from complete state aliasing (e.g., the Blind FSM environment) to a fully observable environment.

To be successful in the Blind FSM or OSFSM environments, an agent must map *sequences of experiences* to actions. An experience, in this case, is the agent's sensory input and the action it selected. For example, the agent could learn that if it takes actions $a_1, a_2, ..., a_n$ in sequence and does not reach the goal then it can subsequently reach the goal in a single step by taking action a_{n+1}. Furthermore, the agent could learn that certain sequences of actions have more utility than others regardless of what non-goal state it is currently at. In effect, the agent must de-alias states by using its recent memories as an identifier.

We expect that the agent would need a significant amount of experience in order to achieve this de-aliasing. If the agent visits the same state through two different paths, the agent may be unable to tell that the states are the same and must learn separate policies for each. Overall, the agent may need to learn substantially more policies than the number of states in order to successfully navigate such an environment. Because of this hindrance, being able to incorporate existing knowledge into the agent, such as through knowledge engineering, could significantly improve the speed at which the agent learns.

In previous research, an algorithm called MaRz (Rodriguez et al. 2017) has shown success in Blind FSM environments. MaRz performs an A*-Search (Russell and Norvig 2009) over possible sequences of actions to find the shortest *universal sequence*. A universal sequence is a sequence of actions that causes the agent to reach the goal from any non-goal state, and such a sequence must always exist in an FSM where every state has a path to goal.

However, MaRz cannot incorporate observations into its algorithm, limiting its scope. Furthermore, the best candidate we see for using knowledge engineering in the MaRz algorithm is through the heuristic function for the search. But the method for converting expert domain knowledge into a search heuristic seems non-trivial.

Other algorithms called Near Sequence Memory (NSM), Utile Suffix Memory (USM) and Noisy Utile Suffix Memory (NUSM) have been shown to be effective in these kinds of environments (McCallum 1995b; 1995a; Shani and Brafman 2005). These operate by looking for the closest past experiences to the most recent events in memory. Then, these algorithms determine which action taken in these past experiences was most effective for reaching the goal. Sometimes, the agent will randomly explore instead.

Similarly to MaRz, these algorithms do not have a clear path to integrating expert knowledge. Since all "knowledge" these algorithms learn is stored as chains of memories, it appears expert knowledge would have to be stored in a similar fashion in order for these algorithms to use it. A more convenient algorithm would expect expert knowledge to be in the format of a set of declarative facts.

In contrast to these other algorithms, ARO generates and employs facts similarly to what an expert may provide, making the learned experience of the agent and the experience of an expert interchangeable. To this end, we will define a *rule*, which records a known probabilistic pattern of cause and effect. This more naturally fits what expert knowledge may appear as. For example:

- Performing action a will cause x.
- If you just performed action a_0 and then you saw x, then you can perform action a_1 to complete the task.
- If you see x, then performing action a will cause y or z to occur at random.

All of these examples may be formatted as rules. We will formally define rules later in this paper, but for now it is imperative to more formally define the environments.

Environment

We will now formally classify the environment for ARO. We define the environment by a tuple (M, G, O, f). The machine M is a finite state machine with state set S, alphabet α, and transition function T such that the goal state G is in S and there is a path from every state in S to G. The observation set O is the set of possible observations (different sensor values) where $G \in O$ denotes the observation of reaching the goal state. The observation function $f : S \rightarrow O$ computes what is observed in any state where $f(s) = G$ if and only if $s = G$.

An agent in this environment is told α and G, but is otherwise provided no information about the environment. The agent is initialized in some unknown random state s_0. At any time-step t when the agent is in state s_t, the agent will be told $f(s_t)$. If s_t is not a goal state, the agent must provide an action $a_t \in \alpha$. The agent will then transition to $s_{t+1} = T(s_t, a_t)$. If instead s_t is a goal state, the agent must provide any action, then it will be moved to non-goal state s_{t+1} at random. This will continue until an fixed but unknown number of goals are reached. The objective of the agent is to minimize the total number of time steps.

In the Blind FSM environment, $O = \{\epsilon, G\}$ and $f(s) = \epsilon$ for non-goal states. In the OSFSM environment, $O = \{0, 1, G\}$, with 0 and 1 denoting whether the state is even or odd, respectively, and $f(s)$ returns the parity of s for non-goal states.

ARO Overview

We are now ready to present ARO, an algorithm designed for navigating environments with extreme state aliasing such as Blind FSM and OSFSM.

ARO stores previous events in units called *episodes*. Each episode consists of the tuple (o, a), where o is the observation $f(s)$ and a is the action taken after the observation. In the OSFSM environment, an example episode may be $(1, b)$. In this episode, the agent was in an odd-numbered state and took action b. When the agent reaches the goal, the action is irrelevant, so we denote such an episode simply with "G." In non-goal episodes, we concatenate the tuple for brevity, so the above episode would be denoted "1b".

Rules

Using these episodes, the agent records patterns of cause and effect in a structure called a *rule*. Rules are denoted like this example:

$$1a \rightarrow 1 : 23\%.$$

This rule states that being in an odd state and taking action a "caused" an odd state 23% of the time. The effect part of a rule is always in this same style, but causes may be more complex. For example, consider this rule:

$$0b, 1a \rightarrow 1 : 6\%.$$

This rule states that being in an even state and performing action b, then reaching an odd state and performing action a has caused the agent to reach an odd state 6% of the time.

More formally, a rule consists of three parts: a sequence of episodes of any length known as the *cause sequence*, an observation known as the *effect*, and a probability. The cause sequence of a rule may be empty, which can be interpreted as the probability that transitioning to a new state causes a particular observation. The number of episodes in the cause sequence of a rule is called the *depth* of a rule.

The three examples for rules given in the background section may now be formally stated. We will use $*$ to denote any possible value.

- $*a \to x : 100\%$
- $*a_0, xa_1 \to G : 100\%$
- $xa \to y : 50\%$ and $xa \to z : 50\%$

As we can see from the final example, it is often useful to consider the set of all rules with the same cause sequence. This lets us consider all possible outcomes from taking a particular action. We call such a set of rules a *rule set*.

We used a tree structure to store rules, but this choice does not affect functionality in contrast to similar algorithms that use a tree-based data structure (McCallum 1995a; Shani and Brafman 2005). So rather than explain this tree, we merely define the function GetRule that takes as inputs both a cause sequence and an effect then produces as an output the corresponding rule. We also define the function GetRuleSet that takes a cause sequence as an input and produces the corresponding rule set as an output.

Since the agent does not have access to the underlying FSM, it cannot compute the probabilities for rules exactly. This leaves us with two choices for developing the rules: learned knowledge and expert knowledge. For learned knowledge, the agent estimates the probability using the rate at which the effect follows the cause sequence in its episode history. For example, if $0b$ occurs 10 times in memory and $0b, G$ occurs 3 times in memory, then the agent would use the rule

$$0b \to G : 30\%.$$

Expert knowledge can be used instead of this estimate if it is provided to the agent. GetRule does not distinguish between the source of the rule when providing it to the agent.

As noted by a reviewer of this paper, it may be necessary to treat expert knowledge as only likely to be correct rather than certain to be correct. To accomplish this, expert knowledge can be associated with a weight, with higher weights indicating a higher confidence in the correctness of the rule. In this case, ARO can treat these weights like the frequency of the observation of this pattern. These frequencies can be added to the total frequencies of ARO's observations in order to compute the probability. For instance, say the rule $0b \to 0 : 100\%$ is provided with a weight of 8, but ARO observes the pattern $0a, 1$ twice. Then ARO would use the rules $0b \to 0 : 80\%$ and $0b \to 1 : 20\%$. Thus, over time expert rules would either be confirmed by ARO's observations or replaced with more accurate rules. In a sense, the weight serves as a kind of Bayesian prior of the actual probability of the effect following the cause. The implications of such a system are not fully explored here.

Now that we have a method for storing patterns of cause and effect, we can use this information to predict future outcomes and find the move that minimizes the expected number of steps to goal.

Expected Value

Given a sequence of previous episodes that have occurred and an observation, we can recursively compute the expected number of moves it will take to reach the goal. This is computed under the assumption that at each choice the agent may have in the future, it will make the best possible move. The algorithm is below. (Note: the GetHeuristic function will be defined later in the paper.)

```
 1: function EV(Episode[] $episodes$, Observation $o$)
 2:     $BestSum = \infty$
 3:     $BestMove = \epsilon$
 4:     for $a$ in $\alpha$ do
 5:         $Sum = 0$
 6:         Let $next = (o, a)$ be a new episode.
 7:         Let $nextSeq$ be $episodes$ appended by $next$
 8:         $nextRuleSet$ = GetRuleSet($nextSeq$)
 9:         if $nextRuleSet$ is empty then          // base case
10:             $Sum$ = GetHeuristic( )
11:         end if
12:         for $rule$ in $nextRuleSet$ do
13:             Let $e$ be the effect of $rule$.
14:             Let $p$ be the probability of $rule$.
15:             if $e = G$ then
16:                 $Sum = Sum + p$
17:             else
18:                 $Sum = Sum + p *$ (EV($nextSeq$, $e$) +1)
19:             end if
20:         end for
21:         if $Sum < BestSum$ then
22:             $BestSum = Sum$
23:             $BestMove = a$
24:         end if
25:     end for
26:     return $BestSum$
27: end function
```

This algorithm also computes the move it expects will take the fewest number of steps to reach the goal on average, stored in the $BestMove$ variable. Let GetBestMove be a function that, when given the same inputs as EV, will return the value of the $BestMove$ variable. The action returned by GetBestMove will minimize the expected number of steps to the goal based on what the agent has learned about the environment, so we would expect this would minimize the average number of steps to the goal over the long term. ARO uses this function to determine what action to make.

There are two aspects of this policy that are not yet explained. First, we cannot compute an expected value for a move we have never taken before. This case is dealt with in line 10 of the code with the GetHeuristic function, which we will define below. Second, there are multiple sequences the agent may pass into GetBestMove. For instance, if the most recent episodes are $1a, 0a$ and the last observation was 0, the agent may call GetBestMove($[]$, 0), GetBestMove($[0a]$, 0), or GetBestMove($[1a, 0a]$, 0). Each sequence may produce a different recommendation, and the agent must choose which recommendation to follow. These are the corresponding topics of the next two sections.

Explore Heuristic

Let us first deal with the case where we must calculate an expected value for a rule set containing no rules. This can be equivalently stated as follows: ARO has never seen the episode sequence $episodes$ occur, then observed the observation o, and then performed some action a. Furthermore, no expert has provided knowledge on what they expect to occur in this scenario. Thus, taking action a can be thought of as an "exploration" by the agent. The agent must choose between exploring this new action and exploiting its existing knowledge to reach the goal.

The explore-exploit tradeoff is well studied under the context of the Multi-Armed Bandit problems (Berry and Fristedt 1985) and has several known solutions (Sutton, Barto, and others 1998; Kearns and Singh 2002; Brafman and Tennenholtz 2002). However, these algorithms are not applicable because they assume that the agent always recognizes the identity of the state it has reached. Research with regard to explore-exploit in environments with extreme perceptual aliasing is more difficult. Lacking an established explore-exploit algorithm, ARO takes a simple approach that seems effective.

To compare the value of this exploration to other actions we have taken, we must decide the value of exploring in the units of expected value. It is possible to find an estimate of the expected value by using the rule set corresponding to removing the first episode from $episodes$. However, repeating this process can lead to infinite depth recursion. Rather than creating a suitable base case to make the recursion finite, we instead attempt to estimate the value that this recursion will return. Let p denote the probability of reaching the goal according to the 0-deep rule $\rightarrow G : p$. We note that if we were to continue recursion infinitely, the probability of reaching the goal in any step should converge on p. If we estimate that this probability holds constant on every step, then the number of steps to goal follows a Geometric distribution with probability p. This gives an expected value of $\frac{1}{p}$. To prevent dividing by 0, we add 1 to the number of goals observed when calculating p. Hence, we can now define the heuristic function:

```
function GETHEURISTIC( )
    return $1/p$
end function
```

As a benefit of this technique, we have derived a quantitative value of exploration without using hyperparameters, in contrast to traditional techniques (Sutton, Barto, and others 1998; Kearns and Singh 2002; Brafman and Tennenholtz 2002). Thus, this is particularly suitable to online learning and artificial general intelligence problems where hyperparameter tuning is not desirable. Also of note is that $\frac{1}{p}$ nearly equals the average number of steps to goal over the lifetime of the agent (they are not equal because we must add one to the number of goals). This means that ARO's explore-exploit solution has an intuitive description: explore if the alternative is worse than average.

This intuitive idea appears in other contexts. Notably, an algorithm called POKER has been applied to the multi-armed bandit problem where the agent is not able to test all the arms before a time horizon is reached (Vermorel and Mohri 2005). In this context, the value of an arm is estimated based on the average value of other arms.

Sequence Selection

The agent may now calculate the best move given a sequence of recent episodes. However, different sized sequences may produce different results. If a large sequence is used, the episode sequence is more likely to be rare in memory, producing unreliable predictions. Furthermore, a long sequence may contain superfluous information, which would imply that a more common and shorter sequence would be a sufficient state identifier. With a small sequence, the episode sequence may be too common for the agent to know where it is in the machine or to determine if it is going in a loop, which could cause undesirable behavior such as infinite looping. To resolve this dilemma, ARO uses a greedy strategy of taking the episode sequence that produces the smallest expected value and the smallest such sequence if there are multiple options. Hence, if the agent has knowledge of a fast path to the goal, it will take it regardless of sequence length.

This alone, however, is not sufficient to prevent loops. At timestep t, say the sequence seq has the property EV(seq, o) $= x$, where x is the minimal expected value and o is the current observation. Hence, the action GETBESTMOVE(seq, o) is performed. Let $nextSeq$ be seq appended by the new episode generated by performing that action. We found that the expected value of $nextSeq$ in timestep $t + 1$ can be higher than x, usually from an improbable event informing the agent that it is farther from the goal than the average case. When this occurred, we observed that the agent would often switch to a shorter sequence with a better expected value for a move recommendation. However, this is not because the agent actually knows a faster path to goal, but because the agent effectively "forgets" that it is in a poor position. After the agent has performed a few moves, it will again learn that it is in a poor position and switch to a shorter sequence for a recommendation. By repeating this behavior, the agent gets stuck in a loop.

To deal with this issue, we force the agent to deal with the bad event by not letting the agent switch to a different sequence for a move recommendation. In other words, if seq was used to find the best move at timestep t, then $nextSeq$ must be used to find the best move at timestep $t + 1$. There are two exceptions to this rule:

1. The agent observes a goal. Because the agent is moved randomly upon reaching the goal state, there is no information to be gained by using a sequence with a goal observation.
2. No rules apply to the current situation using $nextSeq$. This implies the agent is in a novel scenario and the current sequence can provide no data to guide the agent.

Using these procedures, we can now fully define the policy of ARO. We define the global variable $prevSeq$, which stores the sequence from the previous time step used to determine which action to take, and is initialized to the empty sequence. ARO's policy is then described in the following algorithm.

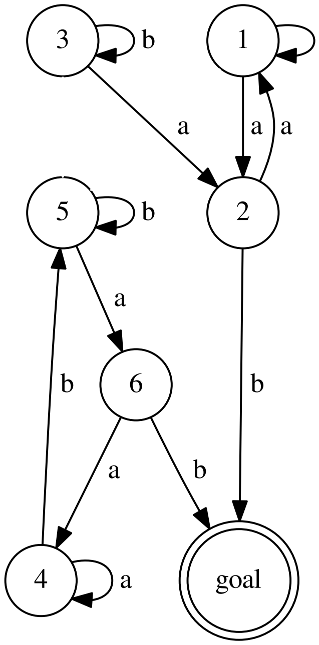

Figure 1: An example FSM

```
function POLICY(Observation $o$)
    if $o = G$ then
        Set $prevSeq$ to the empty sequence.
        return $\alpha[0]$
    end if
    Let $e$ be the last episode.
    Let $seq$ be $prevSeq$ appended by $e$.
    if There are no rules with cause sequence $seq, o*$ or $seq$ contains the episode $G$ then
        Set $prevSeq$ to the sequence of recent episodes such that EV($prevSeq$, $o$) is minimized.
        If there is a tie, use the shortest possible sequence.
    else
        $prevSeq = seq$
    end if
    return GETBESTMOVE($prevSeq$, $o$)
end function
```

Example

Let us use the FSM in Figure 1 as an example for ARO to navigate. Furthermore, suppose that the environment will provide an odd state sensor and that the following two rules are provided by experts on this FSM:

$$\rightarrow 1 : 3/7; 1a, 0b \rightarrow G : 100\%$$

Note that the first rule says odd states happen only $3/7$ths of the time. This is because the goal state should occur $1/7$th of the time and the goal is neither even nor odd. Finally, suppose ARO randomly begins in state 6.

Being in an even state, ARO will be provided an observation of 0. Immediately, ARO will form the new rule $\rightarrow 0 : 4/7$ because ARO has never seen an even state before. This probability is chosen because ARO has seen an even state 100% of the time it was not in an odd state, and based on the rule $\rightarrow 1 : 3/7$, ARO should not be in an odd state $4/7$ths of the time. At this point, ARO has seen 1 state and 0 goals, so $p = \frac{1}{2}$ and our heuristic is 2 moves to the goal. (Recall that we add one to the number of goals when computing p.) ARO will compute an expected value of 2 for moves a and b since it has no rules to predict the outcome of either action. In case of a tie, ARO will arbitrarily pick the earlier move in the alphabet, in this case a. Before it returns, ARO will also update $prevSeq$ to $[0a]$. ARO will then transition to state 4.

ARO again receives an observation of 0. ARO will form the new rule $0a \rightarrow 0 : 100\%$ because of this observation. The heuristic will also update to 3. ARO will look for all rule sets with cause sequence $0a, 0*$ because $[0a]$ is $prevSeq$. However, since no rules apply with this sequence, ARO will check all possible previous sequences to find rule sets. Those sequences are $[]$ and $[0a]$.

First, ARO calls EV($[]$, 0). The move b has no rule set apply and thus has an expected value of 3, the heuristic. The move a will find the rule set consisting solely of the rule $0a \rightarrow 0 : 100\%$, which will cause a recursive call EV($[0a]$, 0). In this call, ARO will find no rule sets that apply and thus return the heuristic of 3 with $BestMove$ set arbitrarily to a. On return, ARO adds 1 to the return value to account for the step of reaching that state and multiplies by $p = 100\%$, so a has an overall expected value of 4. Since EVreturns the minimum value, it will return 3 with $BestMove$ set to b.

Next, from the POLICY function, ARO will try the other sequence by calling EV($[0a]$, 0). Note that we already evaluated the result of this call in the last paragraph - returning 3 with $BestMove$ set to a. This overlap is very common. If EVcalls are cached, it is typically only necessary to call EVa handful of times in order to generate all values needed to select an action. We have also tied for best value since both calls returned 3, so we select the move generated by the shortest sequence, which in this case is the move b. Finally, $prevSeq$ is set to $[0b]$ and ARO transitions to state 5.

ARO will receive an observation of 1, creating the new rules $0b \rightarrow 1 : 100\%$ and $0a, 0b \rightarrow 1 : 100\%$. Again, no rules apply so all sequences are checked. This operation is common when ARO begins to explore but becomes rare once ARO has gathered data. Since the agent has never seen an odd state before, the only helpful rule is the expert rule $1a, 0b \rightarrow G : 100\%$. It may appear that the agent can use this rule, but this is not the case. The agent does not know how likely it is to reach an even state by making the move a; in fact, the agent does not even know if this outcome is possible. As such, it will defer to the heuristic for computing the expected value of a and not use this rule at all. As a consequence of this fact, simpler rules are vastly more important to the agent than complex ones because the former are needed in order to capitalize on the latter. Luckily, those rules are also the easiest for the agent to accurately learn. The best move found will be a as proposed by the empty sequence, so the agent will move into state 6 with $prevSeq$ set to $[1a]$.

At this point, you may notice that ARO is performing a kind of breadth first search. With very little knowledge of how the goal is reached, this is the best strategy ARO can perform. However, once the goal has been reached, ARO will begin biasing its actions toward moves and sequences of moves that are more likely to reach the goal.

Next, ARO receives an observation of 0 and creates the rules $1a \to 0 : 100\%$, $0b, 1a \to 0 : 100\%$, and $0a, 0b, 1a \to 0 : 100\%$. These may seem redundant, but they are all stored independently so that each of their probabilities may change independently in the future. The agent may now capitalize on its expert knowledge. Using $prevSeq$, the agent finds b is the best move with an expected value of 1 because of the rule $1a, 0b \to G : 100\%$. This beats move a, with an expected value the same as the heuristic at 5. ARO then takes move b to reach the goal.

Upon reaching the goal, ARO will update and create several rules. A full list of rules that ARO knows is presented below, with rules of equal depth on the same line:

$$\to 1 : 3/7; \to 0 : 3/7; \to G : 1/7$$

$$0a \to 0 : 100\%; 0b \to 1 : 50\%; 0b \to G : 50\%; 1a \to 0 : 100\%$$

$$0a, 0b \to 1 : 100\%; 0b, 1a \to 0 : 100\%; 1a, 0b \to G : 100\%$$

$$0a, 0b, 1a \to 0 : 100\%; 0b, 1a, 0b \to G : 100\%$$

Even though the agent saw 3 even states and only one odd state, the expert knowledge provided has allowed the 0-deep rules to converge on the correct probabilities. The agent has very little data, so some of the rules it created are incorrect. For instance, $0a \to 0$ should not have a 100% probability due to state 2. But the agent has also correctly discovered patterns such as $0a, 0b \to 1 : 100\%$ which it will be able to immediately use to help navigate the FSM.

Results

To assess ARO, we compared its performance in the OSFSM environment to that of two other algorithms: MaRz (Rodriguez et al. 2017) and Nearest Sequence Memory (McCallum 1995b). Since these agents cannot incorporate knowledge engineering no rules were provided for ARO *a priori*. Since we would expect that being provided expert rules would improve performance, this data can be seen as a worst-case outcome for ARO.

There were a few minor modifications to the implementation of ARO from the above specification to improve memory usage and run time. First, there is no need to record a rule if a prefix of the cause sequence is unique. If the prefix occurs again, we may look back through history to create the rule on the fly. This modification does not affect the behavior of ARO. Second, we cache the result of GETEV and GETBESTMOVE after every goal, which dramatically improves computation time. This technically deviates from the specification because the value of GETHEURISTIC is saved and not recomputed. Since $1/p$ trends downward as the agent learns more, this means that the agent undervalues explores compared to the specification. We found the overall effect on the agent to be small.

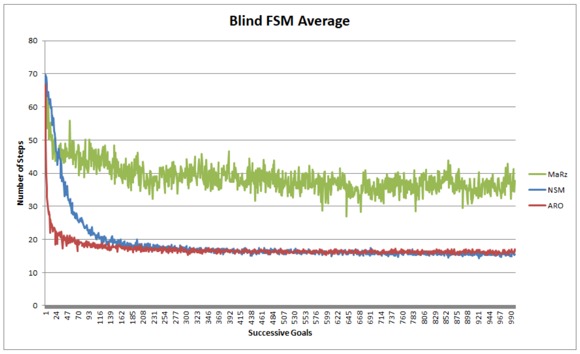

Figure 2: Agents in the Blind FSM Environment with 50 states and an alphabet size of 3. Data averaged from 1000 trials with random FSMs.

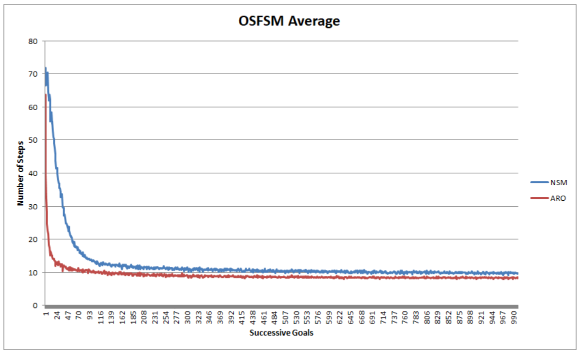

Figure 3: Agents in the Odd-Sensor FSM Environment with 50 states and an alphabet size of 3. Data averaged from 1000 trials with random FSMs.

Figure 2 shows the performance of all three agents in the Blind FSM environment. Figure 3 shows the results for the OSFSM environment. These results are shown for a FSM with 50 states and a three-letter alphabet (three possible actions per state). The abscissa measures successive trips to the goal. The ordinate measures the number of actions required to reach the goal on that trip.

Predictably, the number of steps to reach the goal declines exponentially on successive trips the goal, indicating that the agent is learning to improve its behavior. This is true for all three algorithms. MaRz learns faster than NSM for the first few goals but converges at a much less efficient policy. ARO learns significantly faster than NSM and converges at a policy with nearly identical performance to NSM.

ARO is able to learn near optimal behavior much more rapidly than the other algorithms. These results were consistent across a variety of different machines ranging from 10 to 90 states and alphabets of 2 to 9 letters (not shown).

Future Work

This work demonstrates that ARO is effective in two specific environments with extreme state aliasing. Other tests (not in this paper) have indicated that ARO can handle other deterministic environments as well, including environments with little state aliasing.

The most pressing concern is that it is not obvious how best to generalize ARO to accommodate a stochastic observation function, such as a function that returns 0 or 1 at random. With a random observation function, storing all possible rules can quickly become infeasible in terms of memory because the number of rules with a given depth is exponential and unbounded. This is unlike a deterministic observation function, where the number of rules with a given depth is no greater than the total number of states of the FSM - one rule corresponding to each possible initial state of the agent at the beginning of the cause sequence. Furthermore, ARO has no means of recognizing that the observations it receives are random, so it uses rules trying to predict the value of random observations as a basis to try to reach the goal. These factors cause ARO to perform poorly.

Another potential enhancement is to allow ARO to learn other kinds of rules with more general causes or effects. For example, in our previous examples of rules we used a wildcard character for a rule. While this rule can be provided by an expert, ARO cannot learn such a rule and must instead learn a similar rule for every possible value of the wildcard.

ARO may also need to be adjusted to address environment with these properties:

- a reward function rather than a single goal state
- a non-deterministic FSM
- a dynamic transition table
- continuously valued sensors

ARO's potential extends beyond knowledge integration. As an online algorithm, ARO can take advantage of knowledge immediately and therefore could be a step towards ways in which humans and artificially intelligent agents work collaboratively on a given task in real time.

In addition, the development of ARO builds upon previous research to explore the application of episodic memory for artificial general intelligence (Rodriguez et al. 2017). ARO, therefore, is likely equally applicable in that context. In particular, the lack of hyperparameters means that it can be applied in a general context and does not need to be retuned each time it is used in a new environment.

References

Berry, D. A., and Fristedt, B. 1985. Bandit problems: sequential allocation of experiments (monographs on statistics and applied probability). *London: Chapman and Hall* 5:71–87.

Brafman, R. I., and Tennenholtz, M. 2002. R-max-a general polynomial time algorithm for near-optimal reinforcement learning. *Journal of Machine Learning Research* 3(Oct):213–231.

Chrisman, L. 1992. Reinforcement learning with perceptual aliasing: The perceptual distinctions approach. In *Proceedings of the Tenth National Conference on Artificial Intelligence, AAAI 1992*, 183–188. AAAI Press.

Hopcroft, J. E.; Motwani, R.; and Ullman, J. D. 2006. *Introduction to Automata Theory, Languages, and Computation (3rd Edition)*. Boston, MA, USA: Addison-Wesley Longman Publishing Co., Inc.

Kearns, M., and Singh, S. 2002. Near-optimal reinforcement learning in polynomial time. *Machine learning* 49(2-3):209–232.

McCallum, A. 1995a. Instance-based utile distinctions for reinforcement learning with hidden state. In *ICML*, 387–395.

McCallum, R. A. 1995b. Instance-based state identification for reinforcement learning. In *Advances in Neural Information Processing Systems*, 377–384.

Rodriguez, C.; Marston, G.; Goolkasian, W.; Rosenberg, A.; and Nuxoll, A. 2017. The MaRz algorithm: Towards an artificial general episodic learner. In *International Conference on Artificial General Intelligence*, 154–163. Springer.

Russell, S., and Norvig, P. 2009. *Artificial Intelligence: A Modern Approach*. Upper Saddle River, NJ, USA: Prentice Hall Press, 3rd edition.

Shani, G., and Brafman, R. I. 2005. Resolving perceptual aliasing in the presence of noisy sensors. In *Advances in Neural Information Processing Systems*, 1249–1256.

Sutton, R. S.; Barto, A. G.; et al. 1998. *Introduction to reinforcement learning*. MIT press Cambridge.

Vermorel, J., and Mohri, M. 2005. Multi-armed bandit algorithms and empirical evaluation. In *European conference on machine learning*, 437–448. Springer.