Information Superiority through Intelligent Information Operations

John W. Sheppard
ARINC
2551 Riva Road
Annapolis, MD 21401
jsheppar@arinc.com

Abstract

The information revolution has created the ability for creating tremendous tools to support the warfighter. Tools for collecting, analyzing, and communicating information are being created to improve the efficiency and efficacy of conducting military operations. Unfortunately, the same information revolution has introduced new vulnerabilities whereby adversaries can acquire, exploit, deny, or destroy information needed to support mission objectives. Information warfare has emerged to provide a new wave of warfare in which the focus has shifted from massive destruction of enemy physical assets to surgical attack on information assets. In this paper, we present a framework for supporting information warfare and information operations using advanced techniques from artificial intelligence and control theory. Specifically, we discuss the combination of techniques from approximate reasoning, dynamic programming, and game theory to define a capability to support the conduct of information operations.

Introduction

Information systems have become an essential element of military, government, commercial, and academic operation. The proliferation of computers, networks, and related technologies has provided the capability to rapidly collect, store, analyze, and disseminate information for a variety of purposes. The expediency, efficiency, productivity, and profitability of organizations and individuals have been significantly enhanced by this information revolution. The military especially benefits from this progress by providing decision makers unprecedented quantity, quality, and timeliness of information. The commander with the ability to know the order of battle, analyze events, and distribute critical information possesses a powerful advantage.

The benefits afforded by the information revolution are balanced by some problems. Information is a potent weapon and a lucrative target. The environment in which information is disseminated and stored provides a means for unauthorized access and manipulation. Nations, groups, and individuals seek to acquire, exploit, and protect information in support of their objectives. This exploitation and protection of information can occur for economic and political reasons as well as for military advantage. Strategies, both offensive and defensive, are being formulated to address actions involving the denial, exploitation, corruption, and destruction of enemy information. These strategies form the core of Information Operations (IO) and Information Warfare (IW).

In this paper, we discuss a framework for supporting a C3/C4 analyst in information operations. To understand where such a capability will benefit the analyst, we review the concept of the command and control decision and execution cycle, also known as the "OODA Loop." The OODA loop consists of four distinct phases corresponding, respectively, to *observe*, *orient*, *decide*, and *act* (Figure 1).

The first phase of the OODA loop is the observation phase. At this point, the analyst or command collects information about the battlespace (i.e., environment) within which information operations will occur. Typically, observation is constrained to data collection from sensors and any processing necessary to support the assessment of the information in the next phase. The second phase is orientation. In this phase, the

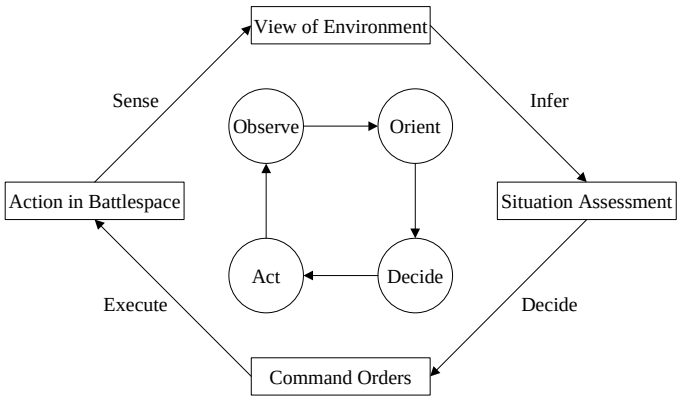

Figure 1. OODA Loop of Command and Control.

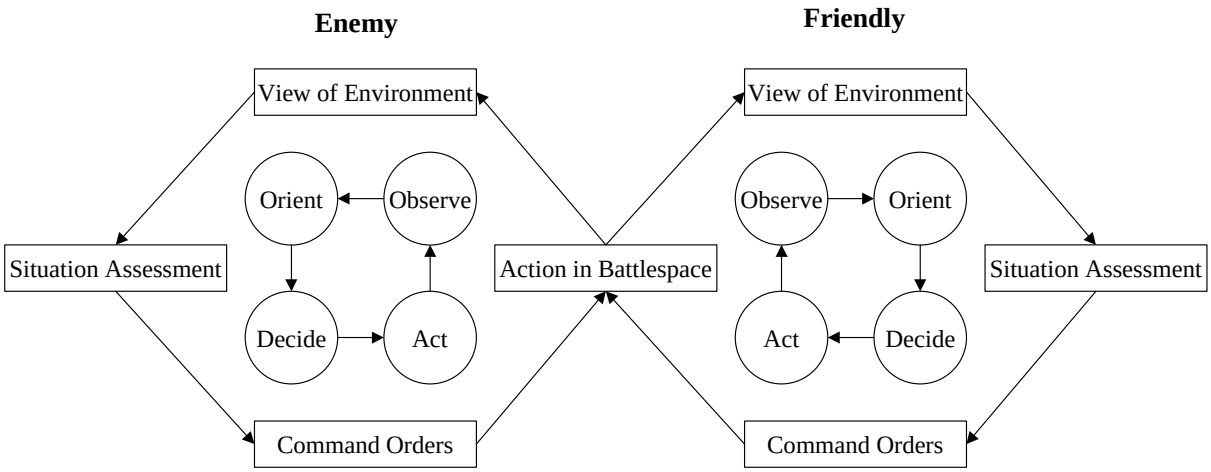

Figure 2. Interacting Decision and Execution Cycles.

analyst or command interprets the information for situation assessment. At this point, inferences are drawn from the information that has been inferred to predict additional attributes about the situation (e.g., risk and strategy). In the third phase, decision, the commander evaluates the results of situation assessment and decides on an appropriate course of action. The resulting orders are passed to those who will execute the orders in the fourth phase, action.

Tools supporting the IO process must work within the decision and execution cycle, as shown in the OODA Loop. Specifically, through the intelligence process, data and information is collected about the target system and the processes that use that system. The intelligence process is responsible for collecting information both in support of planning and real-time execution. Therefore, the information to be observed must be stored in a database or made available as collected. Next, the tool needs to be able to query relevant information related to the current state and infer situation attributes as describe above. Several techniques exist for performing such inference, and we will suggest a particular approach later in this paper. Given the inferred situation, the tools must be capable of assessing the available options in the light of the intended goal, the confidence in the current view of the environment, and the expected utility of executing any of the options. Finally, resulting actions must be reflected in the view of the environment, either through prediction of impact or through the collection of additional information (or both).

To ensure accurate representation and analysis of opponent capabilities in supporting the IO decision process, the opponent's corresponding decision process should be included. This can be represented as an interaction of two OODA loops (Figure 2). Since both cycles affect the environment, the friendly decision process should take into account the enemy's decision cycle to predict expected outcome. This results in the interacting decision cycles being represented as a "game," and techniques from game analysis need to be incorporated into the decision aid.

In a game, three major processes take place that coincide with the OOD phases of the OODA Loop (Figure 3). First, data and information are collected from the environment about the target or opponent. This data is used to capture a current "state" of the game and is combined with previously collected data and information to characterize the entire environment. Such characterization may consist of drawing inferences from known information to estimate or predict unknown attributes of the environment. The combination of known and inferred information defines the current "belief state" of the game. The belief state is used, in combination with a specific objective, to select a course of action for achieving the objective. Once the action is taken, the state of the environment changes, and the process repeats.

The architecture will allow analyst-in-the-loop reasoning to be conducted during the inference process. The intelligence analyst's experience and estimates concerning system architectures, topologies, and configurations, about which little data about the actual target may be available, will be captured and used to build appropriate portions of the Bayesian network. The analyst can then examine the results, query the system for explanations of why certain conclusions were reached, and modify the initial estimates, if necessary. The capability for performing sensitivity analysis will guide the intelligence analyst in determining which parameters are most important for further refinement and research.

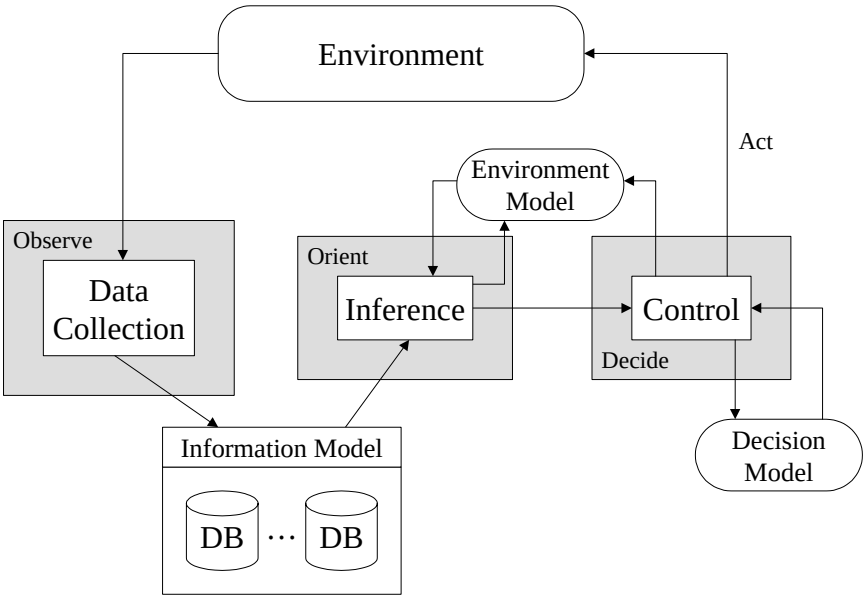

Figure 3. Intelligent IO Process Flow

A Control-Theory View of Information Operations

In general, the information operations (and the OODA loop) can be viewed as a special form of feedback (or closed loop) control where desired network states are obtained by modifying control variables given the current state (Atekson, *et al.*, 1997). Typically, control systems are modeled in one of two ways—through forward models or through inverse models. A forward model uses the current state and the actions that can be applied in that state to predict the results of the actions (i.e., the next state). Typically, this is represented as $s(t+1) = f(s(t),a)$. An inverse model, on the other hand, provides an action given the current state and the desired "outcome," which may be the next state. Thus, the inverse model can be represented as $a = f(s(t),s(t+1))$.

Alternatively, rather than using the next state as an explicit parameter in the model, an expected payoff, ρ, (e.g., percent denial) can be used in the models. Then the forward model becomes $\rho = f(s(t),a)$ and the inverse model becomes $a = f(s(t),\rho)$. Using this alternative form, the feedback control problem can be posed as the problem of optimizing the expected payoff for the controller.

The controller contains a "model" of the process being controlled. This may be an explicit model (e.g., a set of differential equations) or an implicit model (e.g., a neural network or lookup table matching features to actions). In the context of intelligent control, it is expected that the controller will process and modify an implicit model since such a model is both computationally efficient and relatively easy to modify based on past experience.

Once the controller determines the proper action to take (based on a control policy that is either stored or computed), the action is translated into appropriate commands or signals for the actuators that interact with the environment. In the following sections, we will discuss one possible framework using neural networks for implementing this architecture.

Markov Decision Processes

The most common form of representation for the types of decision problems as outlined above are *Markov Decision Processes* (Barto, *et al.*, 1995). A Markov decision process (MDP) is defined by a set of states, $\boldsymbol{S}$, a set of actions, $\boldsymbol{A}$, a set of transitions between states, $\boldsymbol{T}$, associated with a particular action, and a set of discrete probability distributions, $\boldsymbol{P}$, over the set $\boldsymbol{S}$. Thus $\boldsymbol{T} : \boldsymbol{S} \times \boldsymbol{A} \rightarrow \boldsymbol{P}$. Associated with each action while in a given state is a cost (or reward), $c(s,a)$. Given a Markov decision process, the goal is to determine a policy, $\pi(s)$, (i.e., a set of actions to be applied from a given state) to minimize total expected discounted cost. Figure 4 provides a graphical view of an MDP.

Let $f^{\pi}(s_i)$ represent the total expected discounted infinite horizon cost under policy π from state s_i. Let γ ($0 \le \gamma \le 1$) be a discount factor, which has the effect of controlling the influence of future cost on π. Then,

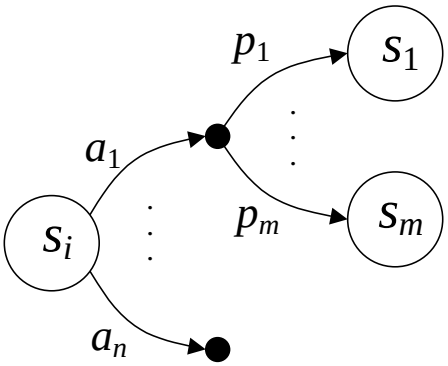

Figure 4. Markov decision process.

$$f^{\pi}(s_i) = E_{\pi}\left[\sum_{t=0}^{\infty} \gamma^t c(s_t, \pi(s_t)) \mid s_0 = s_i\right]$$

where $E_\pi[\bullet]$ is the expectation given policy π, and $c(s,a)$ is the cost of applying action a in state s. Note we can estimate $f^{\pi}(s_i)$ for some $\pi(s_i) = a$ as follows:

$$f^{\pi}(s_i) \approx Q^f(s_i a) = c(s_i, a) + \gamma \sum_{s_j \in \mathbf{S}} p(s_j \mid s_i, a) f(s_j)$$

From this, we are able to establish a policy, π, based on the current estimate Q^f; namely, select $\pi(s_i) = a$ such that,

$$Q^{f_i}(s_i, \pi(s_i)) = \min_{a \in \mathbf{A}} Q^f(s_i, a).$$

This equation is in the form of the *Bellman optimality equation* which can be solved for $f(s_j)$ using several techniques such as dynamic programming (Bellman, 1957).

With the combined OODA loop as depicted in Figure 2, we can generalize the development of a policy within the context of *Markov games* (Sheppard, 1997; Sheppard, 1998). A Markov game is an extension of the MDP in which decisions by multiple players must be considered, and these decisions generally conflict. Under the restriction of two-person games, we define $\boldsymbol{S}$ to be a set of states, $\boldsymbol{A}_1$ and $\boldsymbol{A}_2$ to be sets of actions for players 1 and 2 respectively, $\boldsymbol{T}$ to be a set of transitions similar to the MBP such that $\boldsymbol{T} : \boldsymbol{S} \times \boldsymbol{A}_1 \times \boldsymbol{A}_2 \rightarrow \boldsymbol{P}$. Associated with each player is a cost (or reward) function, $c_1(s,a_1,a_2)$ and $c_2(s,a_1,a_2)$. From the context of IO, the objective is to find a policy $\pi_1(s)$ that maximizes total expected discoutned reward in the presence of an opposing policy $\pi_2(s)$. Value functions for each player analogous to the MDP case can be determined. For alternating games (which is unlikely), policies can be determined for each player given their value functions using minimax. In the event simultaneous games are being played, mixed strategies may be required. For zero-sum games, policies at individual states can be determined using linear programming. For non-zero-sum games (which would result when the value functions for the two players are not complementary), a linear complementarity problem can be constructed and solved using various numeric techniques such as the Lemke-Howson algorithm (von Stengel, 1998).

Given the large state space of the IO scenario, it is likely that a traditional approach using dynamic programming to solve these MDPs will be infeasible. As a result, some form of function approximation will be required for generalizing from representative state-action pairs to the full range of state-action possibilities. One of the more common approaches to function approximation is the use of feed-forward neural networks.

The traditional feed-forward neural network calculates the output of a given node, O_j as $O_j = \sum_{i=1}^{n} w_{ji} x_i$, where n is the number of inputs to the current node. Learning consists of modifying the weights, w_{ji} in such a way to reduce the network error (calculated as $E = \frac{1}{2}(z - O)^2$, where z is the expected network output and O is the actual network output). This weight update (called backpropagation) is accomplished by determining the gradient of this error surface and modifying the weights in the direction of the gradient. Specifically, the weight update rule for backpropagation can be represented as $\Delta w_{ji} = \alpha(z - O)\nabla_w O$ (Rumelhart *et al.*, 1986).

The standard backpropagation algorithm, while performing well on classification tasks, has been shown to have difficulties solving highly dynamic problems such as control problems. In response to these difficulties, work in reinforcement learning and neural networks resulted in the development of a class of algorithms capable of solving specific types of control problems. In particular, *temporal difference* algorithms have been shown to solve highly complex control problems that are posed as MDPs.

Rich Sutton developed an algorithm for training feed-forward neural networks to solve control tasks that can be modeled as an MDP (Sutton, 1988). Sutton's *temporal difference* method focuses on the problem of predicting expected discounted payoff from a given state. This method is applied in "multi-step prediction problems" where payoff is not awarded until several steps after a prediction for payoff is made. At each step, the controller predicts what its future payoff will be, based on several available actions, and chooses its action based on that prediction. However, the ramifications for taking the sequence of actions are not revealed until (typically) the end of the process.

According to Sutton, the temporal difference method is an extension of the prototypical supervised

learning rule that is based on gradient descent (as described above). If we assume a prediction depends upon a vector of modifiable weights *w*, and a vector of state variables *s*, then supervised learning uses a set of paired state vectors and actual outcomes to modify the weights to reduce the error between the predictions and the known outcomes.

The standard, supervised learning method works best for single-step prediction problems. For multi-step prediction, the vector *w* cannot be updated until the end of the sequence, and all observations and predictions must be remembered until the end of the sequence. Sutton's temporal difference method permits incremental update and is based on the observation that

$$z - P_t = \sum_{k=t}^{m} (P_{k+1} - P_k)$$

where P_t is the predicted payoff at time *t*, *m* is the number of steps in the sequence, and $P_{m+1} = z$. In this case, the supervised learning rule becomes

$$\Delta w_{ji}^t = \alpha (P_{t+1} - P_t) \sum_{k=1}^{t} \nabla_w P_k \ .$$

This update can be computed incrementally because it depends only on a pair of successive predictions (P_t and P_{t+1}) and on the sum of past values for $\nabla_w P_t$.

Sutton goes on to describe a family of temporal difference methods based on the influence past updates have on the current update of the weight vector. These methods are based on a parameter, $\lambda \in [0,1]$, which specifies a discount factor in the prediction equation. Sutton refers to this family of equations as the TD(λ) family. When $\lambda = 0$, past updates have no influence on the current update. When $\lambda = 1$, all past predictions receive equal weight. Assuming it is desirable for the update procedure to be more sensitive to recent predictions than to distant predictions, the changes are weighted according to λ^k. Thus the update equation becomes

$$\Delta w_{ji}^t = \alpha (P_{t+1} - P_t) \sum_{k=1}^{t} \lambda^{t-k} \nabla_w P_k \ .$$

Note this equation (and the original gradient descent equation) assumes a single linear combination of weights. This means that for a function to be learned, that function must itself be linear in the inputs (i.e., underlying concepts must be linearly separable). We will address this limitation in the next section and provide a method, based on the "generalized delta-rule (Rumelhart *et al.*, 1986)," for removing this limitation from the proposed approach.

Learning an Optimal Policy

There are two significant questions that must be answered to apply the TD(λ) approach to train a neural network-based controller. First, what is the architecture of the network? Second, given the problem to be solved is likely to be highly non-linear, thus forcing the architecture to be multi-layer, how must the TD(λ) algorithm be modified to work with multi-layer networks?

As indicated in the previous section, a single layer of linear weights feeding an output layer is insufficient to approximate non-linearly separable functions. The most prevalent approach to overcoming this limitation is to permit multiple layers of nodes in the network. Further, multi-layer networks typically apply a transfer function other than a simple weighted sum of the inputs. Specifically, the most commonly used transfer function is a sigmoid function such as the logistic function:

$$y = \frac{1}{1 + e^{-\sum_i w_i x_i}} \ .$$

When such a function is used, the derivative with respect to the weights, $\frac{\partial y}{\partial w}$, is simply $y(1-y)$. Thus, at the output level, the weight update rule becomes

$$\Delta w_{ji} = \alpha (z - O_i) O_i (1 - O_i) x_{ji}$$

where O_i is the output of node *i*, and x_{ji} is the *j*th input to node *i*. If we decide not to use a sigmoid transfer function, then the update rule becomes

$$\Delta w_{ji} = \alpha (z - O_i) x_{ji} \ .$$

Updating the weights of the connections to internal nodes (also called "hidden" nodes) requires one to consider the relative contribution of error from successor nodes. At a hidden node, the update rule is determined as follows. First let

$$\delta_j = O_j (1 - O_j) \sum_{k \in Downstream(j)} \delta_k w_k \ .$$

Then $\Delta w_{ji} = \alpha \delta_j x_{ji}$. For the base case (i.e., when needing to determine δ for an output node), set $\delta_j = O_j (1 - O_j)(z - O_j)$.

We are now prepared to derive the actual temporal difference learning rule to be applied to a neural network representing the information operations problem. At the output layer, we recall from section 2 that we can modify the weights as

$$\Delta w_{ji}^{t} = \alpha(P_{t+1} - P_t)\sum_{k=1}^{t} \lambda^{t-k} \nabla_w P_k \ .$$

This is readily implemented as

$$\Delta w_{ji}^{t} = \alpha(r_t + \lambda P_{t+1} - P_t)\nabla_w P_t$$

where r_t is the immediate payoff received at time t for applying the given action in the current state. Note in this equation, P_t corresponds to the current prediction (i.e., the prediction at time t), and $r_t + \lambda P_{t+1}$ corresponds to the prediction at the next time step (i.e., the prediction at time $t + 1$). If the outputs are linear, the gradient is simply the input value, so the weight update rule reduces to

$$\Delta w_{ji}^{t} = \alpha(r_t + \lambda P_{t+1} - P_t)x_{ji}$$

If one wishes to use a sigmoid activation function at the output, then the weight update rule becomes

$$\Delta w_{ji}^{t} = \alpha(r_t + \lambda P_{t+1} - P_t)P_t(1 - P_t)x_{ji} \ .$$

For the hidden nodes, we still have sigmoid units. Further, the discount factor is not needed since it has been incorporated directly into the error measure at the output. Thus, the weight update rule for the hidden layers remains unchanged from the standard backpropagation learning rule.

Bayesian Networks

Given a neural network for computing a value function, we need a method of representing the state of the control problem being solved. For the IO problem, we suggest using a Bayesian network to capture the current belief in the state of the network under attack. A Bayesian network is a network where the nodes correspond to random variables and directed edges correspond to dependence (i.e., causal) relationships between the random variables (Pearl, 1988).

Within the context of IO, a node in the network will correspond to some attribute of the network (e.g., number and types of radios in a particular subnet, location of the subnet, and type of transmission scheme). Expected values for these attributes are derived from known attributes of the network (obtained through intelligence sources) and conditional probabilities of other values given certain known values within the network. Using basic operations from probability theory, given a Bayesian network and certain known data, probabilities can be propagated through the network to derive expectations for unknown attributes of the network.

Bayesian networks are constructed such that the "roots" of the network (defined to be those nodes that are conditioned on no other random variables) have "prior" probabilities associated with them. Interior nodes of the network have conditional probability tables associated with them indicating the probability of the variable taking on some value given a value of the ancestor nodes. In addition, the networks are constructed to be "acyclic" (i.e., no path exists through the network from a node back to the same node).

Combining Bayesian Networks and Decision Theory

Key to determining a policy that solves a particular MDP is the proper representation of the state of the process. The IO scenario assumes the decision process corresponds to controlling the state of the environment until it reaches some desired state maximizing a particular objective function. Figure 5 provides a graphical view of the approach.

From this figure, we see that we are using the Bayesian network to capture the state of the network. From the beginning of the attack scenario, we establish a "baseline" state using intelligence data, likely network topologies, and likely mission scenarios. Key attributes will be derived from the Bayesian network to form the actual state description for the Markov decision process. Note that this state need not represent the state of the entire network. Such a state representation would be too massive to be able to process efficiently. Rather, the state representation will focus on the area of the network of interest to the attacker (e.g., the local subnet and the portion of the internet to be disrupted or accessed).

From the Bayesian network, an estimate of the current state will be formulated. Based on that state and a set of objectives to be achieved, feasible actions will be

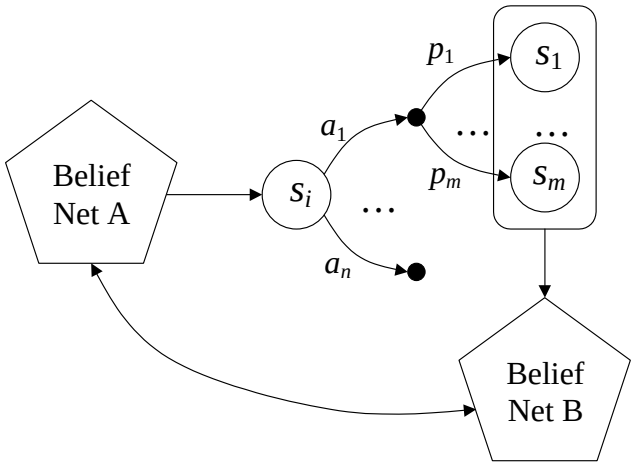

Figure 5. Using Bayesian networks with POMDPs.

considered. The action selected will be one to maximize the ability to achieve the desired objective. Taking this action will alter the state of the network. In the simplest case, the state change will correspond to a modification of the beliefs associated with the random variables within the Bayesian net. In more extreme cases, the change in state of the network may force a change in the structure of the Bayesian net, thus requiring recomputation of the beliefs. Either way, the resulting state is used to select the next action, and the process continues iteratively.

Modeling Partial Observability with Bayesian Networks

Due to the large state space and the probabilistic view on whether or not certain features hold for a given scenario, the decision problem posed by information operations corresponds to a partially observable MDP (POMDP). A POMDP is defined by a set of states ***S***, a set of actions, ***A***, a set of transitions between states associated with a particular actions, ***T***, a set of probability distributions, ***P***, over the set ***S***, a cost function $c(s,a)$, a set of observations, ***Z***, and a set of probability distributions, ***O*** over the set ***Z***. The probability distributions, ***P***, determine the probability of transitioning from state s to state s', given action a. The probability distributions, ***O***, determine the probability of observing z in state s' after taking action a.

In the context of IO, since the current state is captured by value assignments for known random variables and probabilities associated with possible value assignments for unknown random variables, the underlying decision process is partially observable. Key to addressing partial observability is the concept of a belief state (Kaelbling, Littman, & Cassandra 1998). A belief state is defined to be a probability distribution over the set of states, ***S***. Letting $b(s)$ denote the probability assigned to world state s by belief state b, the belief state can be updated using a model of the world (as defined by the POMDP) as follows:

$$\begin{aligned} b'(s') &= \Pr(s' \mid z,a,b) \\ &= \frac{\Pr(z \mid s',a,b)\Pr(s' \mid a,b)}{\Pr(z \mid a,b)} \\ &= \frac{\Pr(z \mid s',a)\sum_{s\in \mathbf{S}} \Pr(s' \mid a,b,s)\Pr(s \mid a,b)}{\Pr(z \mid a,b)} \\ &= \frac{O(s',a,z)\sum_{s\in \mathbf{S}} P(s,a,s')b(s)}{\Pr(z \mid a,b)} \end{aligned}$$

where $\Pr(z \mid a, b)$ can be treated as a normalizer. Note, in the case where the state estimation is given by a Bayesian network, the belief update process will correspond to propagating evidence through the network to revise the specific beliefs of the random variables.

Given a representation for a belief state, the POMDP can be cast as a continuous state-space MDP as follows. Define this "belief" MDP to have a set of belief states, ***B***, a set of actions, ***A*** (as before), a cost function, $\chi(b,a) = \sum_{s\in \mathbf{S}} b(s)c(s,a)$, and a set of transition probabilities defined by

$$\begin{aligned} \tau(b,a,b') &= \Pr(b' \mid a,b) \\ &= \sum_{z\in \mathbf{Z}} \Pr(b' \mid a,b,z)\Pr(z \mid a,b) \end{aligned}$$

where

$$\Pr(b' \mid b,a,z) = \begin{cases} 1; \text{if } \mathrm{BN}(b,a,z) = b' \\ 0; \text{otherwise} \end{cases}$$

for the belief net represented by BN(b, a, z). Since the belief captures all information known about the state so far, even if the underlying decision process is non-Markovian, the modified decision process utilizing these belief states is Markovian. Consequently, any technique for solving continuous state-space MDPs can now be applied (such as the temporal difference approach described earlier).

At this point, the major issue becomes representation of the belief state. Specifically, two issues must be addressed: the dimensionality of the state space and the continuous nature of the belief space. Considering the dimensionality problem, a naïve approach would involve directly mapping the random variables and the associated probabilities of their values to a belief state vector. For example, consider the simple Bayesian network given in Figure 6. In this figure, we see five random variables. For simplicity, assume each variable is a Boolean variable (i.e., either true or false). Assume we have knowledge about nodes A and B that enable us to derive probabilities for C, D, and E. Then the belief state could be represented in one of three ways.

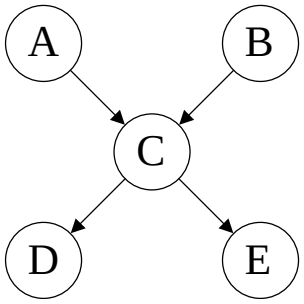

Figure 6. Simple Bayesian network.

1. {Pr(A), Pr(B), Pr(C), Pr(¬C), Pr(D), Pr(¬D), Pr(E), Pr(¬E)}
2. {arg max{Pr(A)}, arg max{Pr(B)}, arg max{Pr(C)}, arg max{Pr(D)}, arg max{Pr(E)}}
3. {⟨arg max{Pr(A)}, Pr(A)⟩, ⟨arg max{Pr(B)}, Pr(B)⟩, ⟨arg max{Pr(C)}, Pr(C)⟩, ⟨arg max{Pr(D)}, Pr(D)⟩, ⟨arg max{Pr(E)}, Pr(E)⟩}

The first form simply represents the probabilities of each of the values for each of the random variables. The second assigns a vector based on the Bayes decision criterion (i.e, select the value with the maximum probability). The third is the same as the second except that it also includes the probabilities.

Note that the first representation is the most explicit and, thereby, the most complex. The second representation is much simpler in that it is no longer infinite; however, a significant amount of information is lost by discarding the probabilities. The third representation provides a compromise between the first and second; however, this representation does not simplify the state space. It seems clear that some form of compaction is required such as the feature selection methods discussed in the previous section.

The second issue focuses on the problem of a continuous state space. Kaelbling, Littman, and Cassandra (1998) address this issue using their "witness" algorithm to construct policy trees based on dominance properties of the underlying policies. Unfortunately, even with clever approaches to pruning the space of policy trees, the approach still requires time exponential in the size of the observation space. Another approach involves constructing a Bayesian network between belief states and representing the conditional probability tables and reward functions using decision trees (Boutilier and Poole 1996). Value iteration is then applied to the decision trees to learn the optimal value functions and is, again, computationally intractable.

The reason for the computational complexity is that both of these approaches focus on computing an exact solution to the POMDP. Substantial savings can occur, however, when settling for an approximate solution. As already discussed, one of the most successful approximate solution methods for MDPs in high-dimensional state spaces is the temporal difference neural network (Tesauro 1992).

Generality of the Framework

Learning approaches such as those described in the reinforcement learning community are called "model-free" because they require no specific model of the underlying Markov decision process. In some ways, this leads to added complexity in that the model must be learned from experience. On the other hand, this assumption provides tremendous power in the ability to adapt the process if network components, sensors, or attack tactics change. Specifically, the details of the control elements are abstracted out of the control model; therefore, it is a simple matter to replace the controller (i.e., the network) with a new controller should the problem change. A neural network can be represented entirely by data (as a set of matrices of weights). Thus no software modification would be required except in mapping inputs and outputs to the appropriate nodes in the network.

Suppose the environment changes but the inputs and outputs remain the same. The only difference to the controller will be the feedback signal (i.e., the payoff) from the environment. Presumably, the new environment will not yield significantly different signals unless there is either a radical change in the task to be performed. In any event, the temporal difference method will accept the new feedback signal and begin to modify its model of the environment immediately.

Adaptation becomes more complicated if the inputs or the outputs change. Since the impact is similar, we will treat both of these situations together. When using a neural network, both the input data and the control data being recommended are represented by numerical input/output in the network. Changes mean that the inputs/outputs must be modified either through inserting a new node, deleting an existing node, or changing a node. Note that changing a node is analogous to a deletion followed by an insertion. If a change is of a similar type, it might make sense to use the original weights as a starting point; otherwise, the weights can be reinitialized for the new node. In all cases, it is probably prudent to retrain the network in the simulated environment before hosting in the controller. The advantage to this iterative approach is that it can bootstrap off of previously learned information.

Conclusion

Overall, the framework described in this paper is very flexible and powerful. It is flexible in its ability to abstract needed information from the environment and in its ability to be encapsulated from the environment. It is powerful in that it supports a wide variety of capabilities including feature extraction, function approximation, and adaptive control. In this paper, we discussed several different algorithms for each of these. The algorithms are not the only ones possible and are offered as representative examples rather than design decisions. In the end, however, it is felt that adaptive approaches such as those offered above will offer superior power and flexibility over scripting or static rule-based reasoning.

References

Barto, A., Bradtke, S., and Singh, S. 1995. "Learning to Act Using Real-Time Dynamic Programming," *Artificial Intelligence*, Special Volume: Computational Research on Interaction and Agency, 72(1): 81–138.

Bellman, R. 1957. *Dynamic Programming*, Princeton, NJ: Princeton University Press.

Boutilier, C., and Poole, D. 1996. "Computing Optimal Policies for Partially Observable Decision Processes Using Compact Representations," *Proceedings of the 13th National Conference on Artificial Intelligence*, AAAI Press, pp. 1168–1175.

Kaelbling, L., Littman, M., and Cassandra, A. 1998. "Planning and Acting in Partially Observable Stochastic Domains," *Artificial Intelligence*, to appear.

Pearl, J. 1988. *Probabilistic Reasoning in Intelligent Systems*, San Mateo, CA: Morgan Kaufmann.

Rumelhart, D., Hinton, G., and Williams, R. 1986. "Learning Internal Representations by Error Propagation," *Parallel Distributed Processing: Explorations in the Microstructures of Cognition*, Vol. 1, D Rumelhart and J. McClelland (Eds.), Cambridge, MA: The MIT Press, pp. 318–362.

Sheppard, J. 1997. *Multi-Agent Reinforcement Learning in Markov Games*, Ph.D. Dissertation, Department of Computer Science, The Johns Hopkins University, Baltimore, MD.

Sheppard, J. 1998. "Co-Learning in Differential Games," *Machine Learning*, special issue on multi-agent learning, Vol. 33, No. 2/3, pp. 201–233.

Sutton, R. 1988. "Learning to Predict by Methods of Temporal Differences," *Machine Learning*, 3:9–44.

Tesauro, G. 1992. "Practical Issues in Temporal Difference Learning," *Machine Learning*, Vol. 8, pp. 257–277.

Von Stengel, B. 1998. "Computing Equilibria for Two-Person Games," *Handbook of Game Theory*, R. Aumann and S. Hart (eds.) Vol. 3, North-Holland, Amsterdam.

Wettschereck, D., Aha, D., and Mohri, T. 1997. "A Review and Empirical Evaluation of Feature Weighting Methods for a Class of Learning Algorithms," *Artificial Intelligence Review*, Vol. 11, pp. 273–314.