Combining Reinforcement Learning with Information-State Update Rules[*]

Peter A. Heeman
Center for Spoken Language Understanding
Oregon Health & Science University
Beaverton OR, 97006, USA
heeman@cslu.ogi.edu

Abstract

Reinforcement learning gives a way to learn under what circumstances to perform which actions. However, this approach lacks a formal framework for specifying hand-crafted restrictions, for specifying the effects of the system actions, or for specifying the user simulation. The information state approach, in contrast, allows system and user behavior to be specified as update rules, with preconditions and effects. This approach can be used to specify complex dialogue behavior in a systematic way. We propose combining these two approaches, thus allowing a formal specification of the dialogue behavior, and allowing hand-crafted preconditions, with remaining ones determined via reinforcement learning so as to minimize dialogue cost.

1 Introduction

Two different approaches have become popular for building spoken dialogue systems. The first is the symbolic reasoning approach. Speech actions are defined in a formal logic, in terms of the situations in which they can be applied, and what effect they will have on the speaker's and the listener's mental state (Cohen and Perrault, 1979; Allen and Perrault, 1980). One of these approaches is the *information state* (IS) approach (Larsson and Traum, 2000). The knowledge of the agent is formalized as the state. The IS state is updated by way of *update rules*, which have *preconditions* and *effects*. The preconditions specify what must be true of the state in order to apply the rule. The effects specify how the state changes as a result of applying the rule. At a minimum, two sets of update rules are used: one set, *understanding rules*, specify the effect of an utterance on the agent's state and a second set, *action* rules, specify which speech action can be performed next. For example, a precondition for asking a question is that the agent does not know the answer to the question. An effect of an answer to a question is that the hearer now knows the answer. One problem with this approach is that although necessary preconditions for speech actions are easy to code, there are typically many speech actions that can be applied at any point in a dialogue. Determining which one is the optimal one is a daunting task for the dialogue designer.

The second approach for building spoken dialogue systems is to use *reinforcement learning* (RL) to automatically determine what action to perform in each different dialogue state so as to minimize some cost function (e.g. Walker, 2000; Levin et al., 2000). The problem with this approach, however, is that it lacks the framework of IS to specify the manner in which the internal state is updated. Furthermore, sometimes no preconditions are even specified for the actions, even though they are obvious to the dialogue designer. Thus RL needs to search over a much larger search space, even over dialogue strategies that do not make any sense. This not only substantially slows down the learning procedure, but also increases the chance of being caught in a locally optimal solution, rather than the global optimal. Furthermore, this large search space will limit the complexity of the domains to which RL can be applied.

In this paper, we propose combining IS and RL. IS update rules are formulated for both the system and the simulated user, thus allowing RL to use a rich formalism for specifying complex dialogue processing. The preconditions on the action rules of the system, however, only need to specify the neces-

[*]The author wishes to thank Fan Yang and Michael English for helpful conversations. Funding from the National Science Foundation under grant IIS-0326496 is gratefully acknowledged.

sary preconditions that are obvious to the dialogue designer. Thus, preconditions on the system's actions might not uniquely identify a single action that should be performed in a given state. Instead, RL is used to determine which of the applicable actions minimizes a dialogue cost function.

In the rest of the paper, we first present an example domain. Section 3 gives an overview of applying RL to dialogue strategy and Section 4 gives an overview of IS. Section 5 demonstrates that IS can be used for simulating a dialogue between the system and a user. Section 6 demonstrates how IS can be used with RL. Section 7 gives results on using hand-crafted preconditions specified in the IS update rules to simplify learning dialogue strategies with RL. Section 8 gives concluding comments.

2 Flight Information Application

To illustrate our proposed approach, we use the flight information domain, similar to that of Levin et al. (2000). The goal of the system is to display a short list of flights that meets the user's requirements. The user is assumed to have a flight in mind, in terms of its destination, origin, airline, departure time, and number of stops. The user might be flexible on some of the parameters. It is assumed that the user will not change his or her mind depending on what flights are found.

In this paper, we are focusing on dialogue management issues, and so we use a semantic representation for both the input and output of the system. The system can ask the user the value of parameter p with 'askconstraint p', and the user will answer with 'constrain p v', where v is the user's preferred value of the parameter.[1] The system can ask whether the user is flexible on the values for parameter p with 'askrelax p', and the user will answer with 'relax p a', where a is either 'yes' or 'no'. The system can do a database query, 'dbquery', to determine whether any flights match the current parameters. If no flights exactly match, 'dbquery' will check if any flights match according to the relaxed restrictions, by ignoring parameters that the system knows the user is flexible on. The system can display the found flights with 'output'. It can also quit at any time. A sample dialogue is given in Fig. 1.

```
system   askconstraint from
user     constrain from miami
system   askconstraint to
user     constrain to sacramento
system   askconstraint departure
user     constrain departure 6pm
system   dbquery miami sacremento - 6pm
system   askconstraint airline
user     constrain airline united
system   dbquery miami sacremento united ...
system   askrelax departure
user     relax departure yes
system   dbquery miami sacremento united ...
system   output {918 11671 13288}
system   finish
```

Figure 1: Sample dialogue

3 Reinforcement Learning (RL)

Given a set of system actions, a set of states, and a cost function that measures the quality of a dialogue, RL searches for an optimal dialogue policy (Sutton and Barto, 1998; Levin et al., 2000).

Cost Function: The cost function assesses how good a dialogue is: the lower the cost, the better the dialogue. RL uses the cost function to provide feedback in its search for an optimal strategy. The cost function is specified by the dialogue designer, and can take into account any number of factors, typically including dialogue length and solution quality.

System Actions: RL takes as input a finite number of actions, and for each state, learns which action is best to perform. The dialogue designer decides what the actions will be, both in terms of how much to combine into a single action, and how specific each action should be.

State Variables: RL learns what system action to perform in each state. The RL states are defined in terms of a set of state variables: different values for the variables define the different states that can exist. The state variables need to include all information that the dialogue designer thinks will be relevant in determining what action to perform next. Any information that is thought to be irrelevant is excluded in order to keep the search space small.

Transitions: RL treats a dialogue as a succession of states, with actions causing a transition from one state to the next. The transition thus encompasses the effect of the system making the speech act,

[1] In contrast to Levin, over-answering by the user is not allowed. The system also does not have a general greeting, to which the user can answer with any of the flight parameters.

the user's response to the system's speech act, and the system's understanding of the user's response. Hence, the transition incorporates a *user simulation*. In applying RL to dialogue policies, the transition from a state-action pair to the next state is usually modeled as a probability distribution, and is not further decomposed (e.g. Levin et al., 2000).

Policy Exploration: RL searches the space of polices by determining Q for each state-action pair *s*-*a*, which is the minimal cost to get to the final state from state s starting with action a. From the Q values, a policy can be determined: for each state s, choose the action a that has the maximum Q value.

Q is determined in an iterative fashion. The current estimates for Q for each state-action are used to determine the current dialogue policy. The policy, in conjunction with the transition probabilities, are used to produce a dialogue run, which is a sequence of state-action pairs, each pair having an associated cost to get to the next state-action pair. Thus, for a dialogue run, the cost from each state-action pair to the final state can be determined. These costs are used to revise the Q estimates.

To produce a dialogue run, the ϵ-greedy method is often used. In this approach, with probability ϵ, an action other than the action specified by the current policy is chosen. This helps ensure that new estimates are obtained for all state-action pairs, not just ones in the current policy. Typically, a number of dialogue runs, an *epoch*, are made before the Q values and dialogue policy are updated. With each successive epoch, a better dialogue policy is used, and thus the Q estimates will approach their true values, which in turn, ensures that the dialogue policy is approaching the optimal one.

3.1 Flight Information Task in RL

To illustrate how RL learns a dialogue policy, we use the flight information task from Section 2.

Actions: The system actions were given in Section 2. The queries for the destination, origin, airline, departure time, number of stops are each viewed as different actions so that RL can reason about the individual parameters. There are also 5 separate queries for checking whether each parameter can be relaxed. There is also a database query to determine which flights match the current parameters. This is included as an RL action, even though it is not to the user, so that RL can decide when it should be performed. There is also an output and a finish action.

State Variables: We use the following variables for the RL state. The variable 'fromP' indicates whether the origin has been given by the user and the variable 'fromR' indicates whether the user has been asked if the origin can be relaxed, and if so, what the answer is. Similar variables are used for the other parameters. The variable 'dbqueried' indicates whether the database has been queried. The variable 'current' indicates whether no new parameters have been given or relaxed since the last database query. The variable 'NData' indicates the number of items that were last returned from the database quantized into 5 groups: none, 1-5, 6-12, 13-30, more than 30). The variable 'outputP' indicates whether any flights have been given to the user. Note that the actual values of the parameters are not included in the state. This helps limit the size of the search space, but precludes the values of the parameters from being used in deciding what action to perform next.

Cost Function: Our cost function is the sum of four components. Each speech action has a cost of 1. A database query has a cost of 2 plus 0.01 for each flight found. Displaying flights to the user costs 0 for 5 or fewer flights, 8 for 12 or fewer flights, 16 for 30 or fewer flights, and 25 for 30 or more flights. The last cost is the solution cost. This cost takes into account whether the user's preferred flight is even in the database, and if so, whether it was shown to the user. The solution cost is zero if appropriate information is given to the user, and 90 points otherwise.

3.2 Related Work in RL

In the work of Levin, Pieraccini, and Eckert (2000), RL was used to choose between all actions. Actions that resulted in infelicitous speech act sequences were allowed, such as asking the value of a parameter that is already known, asking if a parameter can be relaxed when the value of the parameter is not even known, or displaying values when a database query has not yet been performed.

In other work, RL has been used to choose among a subset of the actions in certain states (Walker, 2000; Singh et al., 2002; Scheffler and Young, 2002; English and Heeman, 2005). However, no formal framework is given to specify which actions to choose from.

Furthermore, none of the approaches used a formal specification for updating the RL variables after a speech action, nor for expressing the user simulation. As RL is applied to more complex tasks, with more complex speech actions, this will lead to difficulty in encoding the correct behavior.

Georgila, Henderson, and Lemon (2005) advocated the use of IS to specify the dialogue context for learning user simulations needed in RL. However, they did not combine hand-crafted with learned preconditions, and it is unclear whether they used IS to update the dialogue context,

4 Information State (IS)

IS has been concerned with capturing how to update the state of a dialogue system in order to build advanced dialogue systems (Larsson and Traum, 2000). For example, it has been used to build systems that allow for both system and user initiative, over answering, confirmations, and grounding (e.g. (Bohlin et al., 1999; Matheson et al., 2000)). It uses a set of state variables, whose values are manipulated by update rules, run by a control strategy.

State Variables: The state variables specify the knowledge of the system at any point in the dialogue. This is similar to the RL variables, except that they must contain everything that is needed to completely specify the action that the system should perform, rather than just enough information to choose between competing actions. A number of standard variables are typically used to interface to other modules in the system. The variable 'lastMove' has the semantic representation of what was last said, either by the user or the system and 'lastSpeaker' indicates who spoke the last utterance. Both are read-only. The variable 'nextMove' is set by the action rules to the semantic representation of the next move and 'keepTurn' is set to indicate whether the current speaker will keep the turn to make another utterance.

Update Rules: Update rules have preconditions and effects. The preconditions specify what must be true of the state in order to apply the rule. The effects specify how the state should be updated. In this paper, we will use two types of rules. Understanding rules will be used to update the state to take into account what was just said, by both the user and the system. Action rules determine what the system will say next and whether it will keep the turn.

Control Strategy: The control strategy specifies how the update rules should be processed. In our example, the control strategy specifies that the understanding rules are processed first, and then the action rules if the system has the turn. The control strategy also specifies which rules should be applied: (a) just the first applicable rule, (b) all applicable rules, or (c) randomly choose one of the applicable rules.

Although there is a toolkit available for building IS systems (Larsson and Traum, 2000), we built a simple version in Tcl. Update rules are written using Tcl code, which allows for simple interpretation of the rules. The state is saved as Tcl variables, and thus allows strings, numbers, booleans, and lists.

4.1 Flight Information Example in IS

We now express the flight information system with the IS approach. This allows for a precise formalization of the actions, both the conditions under which they should be performed and their effects.

The IS state variables are similar to the RL ones given in Section 3. Instead of the variable 'fromP', it includes the variable 'from', which has the actual value of the parameter if known, and '' otherwise. The same is true for the destination, airline, departure time, and number of stops. Instead of the RL variable 'NData' and 'outputP, 'results' holds the actual database and 'output' holds the actual flights displayed to the user.

Figure 2 displays the system's understanding rules, which are used to update the state variables after an utterance is said. Although it is common practice in IS to use understanding rules even for one's own utterances, the example application is simple enough to do without this. Understanding rules are thus only used for understanding the user's utterances: giving a parameter value or specifying whether a parameter can be relaxed. As can be seen, any time the user specifies a new parameter or relaxes a parameter, 'current' is set to false.

Figure 3 gives the action rules for the system. Rules for querying the destination, departure, and number of stops are not shown; neither are the rules for querying whether the destination, origin, airline, and number of stops can be relaxed. The effects of the rules show how the state is updated if the rule is applied. For most of the rules, this is simply to set ‘nextMove’ and ‘keepTurn’ appropriately. The ‘dbquery’ action is more complicated: it runs the database query and updates ‘results’. It then updates the variables ‘queriedDB’, and ‘current’ appropriately. Note that the actions ‘dbquery’ and ‘output’ specify that the system wants to keep the turn.

The preconditions of the update rules specify the exact conditions under which the rule can be applied. The preconditions on the understanding rules are straightforward, and simply check the user’s response. The preconditions on the action rules are more complex. We divide the preconditions into the 4 groups given below, both to simplify the discussion of the preconditions, and because we use these groupings in Section 7.

Speech Acts: Some of the preconditions capture the conditions under which the action can be performed felicitously (Cohen and Perrault, 1979; Allen and Perrault, 1980). Only ask the value of a parameter if you do not know its value. Only ask if a parameter can be relaxed if you know the value of the parameter. Only output the data if it is still current and more than one flight was found. These preconditions are labeled as ‘sa’ in Fig. 3.

Application Restrictions: These preconditions enforce the specification of the application. For our application, the system should only output data once: once data is output, the system should end the conversation. These preconditions are labeled as ‘app’ in Fig. 3.

Partial Strategy: These preconditions add addition constraints that seem reasonable: ask the ‘to’, ‘from’, and ‘departure’ parameters first; never relax the ‘to’ and ‘from’; and only ask whether ‘airline’ and ‘stops’ can be relaxed if the database has been queried. Furthermore, the system may only output data if (a) the number of flights is between 1 and 5, or (b) the number of flights is greater than 5 and ‘airline’ and ‘stops’ have both been asked. These preconditions are labeled as ‘ps’ in Fig. 3.

Baseline: The last group of preconditions (together with the previous preconditions) uniquely identify a single action to perform in each state, and

Understand Answer to Constrain Question	
Pre:	[lindex $lastMove 0] == “constrain”
Eff:	set [lindex LastMove 1] [lindex LastMove 2]
	set current 0
Understand Yes Answer to Relax	
Pre:	[lindex lastMove 0] == “relax”
	[lindex lastMove 2] == “yes”
Eff:	set [lindex lastMove 1]R yes
	set current 0
Understand No Answer to Relax	
Pre:	[lindex lastMove 0] == “relax”
	[lindex lastMove 2] == “no”
Eff:	set [lindex lastMove 1]R no

Figure 2: Understanding Rules for System

Ask Origin of Flight		
Pre:	$from == ‘’	sa
	$output == ‘’	app
Eff:	set nextMove “askconstraint from”	
	set keepTurn false	
Ask Airline of Flight		
Pre:	$airline == ‘’	sa
	$output == ‘’	app
	$departure != ‘’	ps
	$queriedDB == true	base
	$current == true	base
	[llength $results] > 5	base
Eff:	set nextMove “askconstraint to”	
	set keepTurn false	
Ask Whether Departure Time can be Relaxed		
Pre:	$departure != ‘’	sa
	$departureR == ‘’	sa
	$output != ‘’	app
	$queriedDB == true	base
	$current == true	base
	$results == {}	base
Eff:	set nextMove ‘askrelax from’	
	set keepTurn false	
Query the Database		
Pre:	$current == false	sa
	$output == ‘’	app
	$departure != ‘’	ps
Eff:	set results [DBQuery $from $to $airline ...]	
	set queriedDB true	
	set current true	
	set nextMove dbquery	
	set keepTurn true	
Output Results to User		
Pre:	$current == true	sa
	$results != {}	sa
	$output == ‘’	app
	[llength $results] < 6 || ([llength $results] > 5 && $airline != “” && $stops != “”)	ps
Eff:	set nextMove “output $results”	
	set output $results	
Finish		
Pre:	$output != ‘”	app
Eff:	set nextMove finish	
Quit		
Pre:	$output == ‘’	app
	$current == true	app
	$results == {}	app
	$airline != ‘’ || $airlineR != ‘’	base
	$stops != ‘’ || $stopsR != ‘’	base
Eff:	set nextMove finish	

Figure 3: Action Rules for System

thus completely specifies a strategy. These are labeled as 'base' in Fig. 3. The strategy that we give is based on the optimal strategy found by Levin et al. (2000). After the system asks the values for the 'from', 'to', and 'departure' variables, it then performs a database query. If there are between 1 and 5 flights found, they are displayed to the user. If there are more than 5, the system asks the value of 'airline' if unknown, otherwise, 'number of stops'. If there are 0 items, it tries to relax one of 'departure', 'airline', and 'stops', in that order (but not 'from' or 'to'). Any time new information is gained, such as a parameter value or a parameter is relaxed, the database is requeried, and the process repeats.

5 Implementing the Simulated User

Normally, with IS, the system is run against an actual user, and so no state variables nor update rules are coded for the user. To allow the combination of IS with RL, we need to produce dialogues between the system and a simulated user. As the IS approach is very general, we will use it for implementing the simulated user as well. In this way, we can code the user simulation with a well-defined formalism, thus allowing complex user behaviors. Hence, two separate IS instantiations will be used: one for the system and one for the user. The system's rules will update the system's state variables, and the user's rules will update the user's state variables; but the two instantiations will be in lock-step with each other.

We built a simulator that runs the system's rules against the user's. The simulator (a) runs the understanding rules for the system and the user on the last utterance; then (b) checks who has the turn, and runs that agent's action rules; and then (c) updates 'lastSpeaker' and 'lastMove'. It repeats these three steps until the 'finish' speech act is seen.

5.1 Flight Information Task

The user has the variables 'from', 'to', 'departure', 'airline', and 'stops', which hold the user's ideal flight, and are set before the dialogue begins. The variables 'fromR', 'toR', 'departureR', 'airlineR', and 'stopsR' are also used, and are also set before the dialogue begins. No other variables are used.

For the flight application, separate update rules are used for the user. There are two types of queries to which the user needs to react, namely, 'askcontraint' and 'askrelax'. This domain is simple enough that we do not need separate understanding and action rules, and so we encompass all reasoning in the action rules, shown in Fig. 4. The first rule is for answering system queries about the value of a parameter. The second is for answering queries about whether a parameter can be relaxed.

Answer Constrain Question
```
Pre:   [lindex $lastMove 0] == "askconstraint"
Eff:   set nextMove "constraint [lindex $lastmove 1]
                [set [lindex $lastmove 1]]"
       set haveTurn 0
```
Answer Relax Question
```
Pre:   [lindex $lastMove 0] == "askrelax"
Eff:   set nextMove "relax [lindex $lastmove 1]
                [set [lindex $lastMove 1]R]"
       set haveTurn 0
```
Figure 4: Action Rules for User

6 Combining IS and RL

RL gives a way to learn the best action to perform in any given state. However, RL lacks a formal framework for specifying (a) the effects of the system's actions, (b) hand-crafted preconditions of the system's actions, and (c) the simulated user. Hence, we combine RL and IS to rectify these deficits. IS update rules are formulated for both the system and the simulated user, as done in Section 5.1. The preconditions on the system's action rules, however, only need to specify a subset of the preconditions, ones that are obvious the dialogue designer. The rest of the preconditions will be determined by RL, so as to minimize a cost function. To combine these two approaches, we need to (a) resolve how the IS and RL state transitions relate to each other; (b) resolve how the IS state relates to the RL state; and (c) specify how utterance costs can be specified in the general framework of IS.

Transitions: When using IS for both the system and user simulation, the state transitions for each are happening in lock-step (Section 5.1). In combining RL and IS, the RL transitions happen at a courser granularity than the IS transitions, and group together everything that happens between two successive system actions. Thus, the RL states are those IS states just before a system action.

State Variables: For the system, we add all of the RL variables to the IS variables, and remove any duplicates. The RL variables are thus a subset of the IS variables. Some of the variables might be simplifications of other variables. For our flight example, we have the exact values of the origin, destination, airline, departure time, and number of stops, as well as a simplification of each that only indicates whether the parameter has been given or not.

Rather than have the system's IS rules update all of the variables, we allow variables to be declared as either *primitive* or *derived*.[2] Only primitive variables are updated by the effects of the update rules. The derived variables are re-computed from the primitive ones each time an update rule is applied. For our flight example, the variables 'fromP', 'toP', 'airlineP', 'departureP', 'stopsP', 'outputP', and 'NData' are derived variables, and these are updated via a procedure.

As the RL variables are a subset of the IS variables, the RL states are coarser than the IS states. We do not allow hand-crafted preconditions in the system's action rules to distinguish at the finer granularity. If they did, we would have an action that is only applicable in part of an RL state, and not the rest of it. However, RL needs to find a single action that will work for the entire RL state, and so that action should not be considered. To prevent such problems, the hand-crafted preconditions can only test the values of the RL variables, and not the full set of IS variables. Hence, we rewrote the preconditions in the action rules of Fig. 3 to use the RL variables. This restriction does not apply to the system's understanding rules, nor to the user rules, as those are not subject to RL.

Cost Function: RL needs to track the costs incurred in the dialogue. Rather than leaving this to be specified in an ad-hoc way, we include state variables to track the components of the cost. This way, each update rule can set them to reflect the cost of the rule. Just as with other interface variables (e.g. 'keepTurn'), these are write-only. For our flight example, the output action computes the cost of displaying flights to the user, and the database query action computes the cost of doing the database lookup.

[2]This same distinction is sometimes used in the planning literature (Poole et al., 1998).

7 Evaluation

To show the usefulness of starting RL with some of the preconditions hand-crafted, we applied RL using four different sets of action schemes. The first set, 'none', includes no preconditions on any of the system's actions. The second through fourth sets correspond to the precondition distinctions in Fig. 3, of 'speech act', 'application' and 'partial strategy'.

For each set of action schemas, we trained 30 dialogue policies using an epoch size of 100. Each dialogue was run with the ϵ-greedy method, with ϵ set at 0.15. After certain epochs, we ran the learned policy 2500 times strictly according to the policy. We found that policies did not always converge. Hence, we trained the policies for each set of preconditions for enough epochs so that the average cost no longer improved. More work is needed to investigate this issue.

The results of the simulations are given in Table 1. The first row reports the average dialogue cost that the 30 learned policies achieved. We see that all four conditions achieved an average cost less than the baseline strategy of Fig. 3, which was 17.17. The best result was achieved by the 'application' preconditions. This is probably because 'partial' included some constraints that were not optimal, while the search strategy was not adequate to deal with the large search space in 'speech acts' and 'none'.

The more important result is in the second row of Table 1. The more constrained precondition sets result in significantly fewer states being explored, ranging from 275 for the 'partial' preconditions, up to 18,206 for no preconditions. In terms of number of potential policies explored (computed as the product of the number of actions explored in each state), this ranges from 10^{58} to 10^{7931}. As can be seen, by placing restrictions on the system actions, the space that needs to be explored is substantially reduced.

The restriction in the size of the search space affects how quickly RL takes to find a good solution. Figure 5 shows how the average cost for each set of

	None	SA	App.	Partial
Dialogue Cost	16.65	16.95	15.24	15.68
States Explored	18206	5261	4080	275
Policies ($\log_{10}$)	7931	2008	1380	58.7

Table 1: Comparison of Preconditions

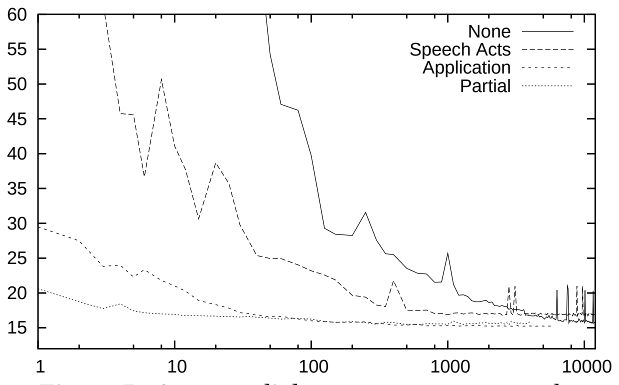

Figure 5: Average dialogue cost versus epochs

preconditions improved with the number of epochs. As can be seen, by including more preconditions in the action definitions, RL is able to find a good solution more quickly. For the 'partial' preconditions, after 10 epochs, RL achieves a cost less than 17.0. For the 'application' setting, this does not happen until 40 epochs. For 'speech act', it takes 1000 epochs, and for 'none', it takes 3700 epochs. So, adding hand-crafted preconditions allows RL to converge more quickly.

8 Conclusion

In this paper, we demonstrated how RL and IS can be combined. From the RL standpoint, this allows the rich formalism of IS update rules to be used for formalizing the effects of the system's speech actions, and for formalizing the user simulation, thus enabling RL to be applied to domains that require complex dialogue processing. Second, use of IS allows obvious preconditions to be easily formulated, thus allowing RL to search a much smaller space of policies, which enables it to converge more quickly to the optimal policy. This should also enable RL to be applied to complex domains with large numbers of states and actions.

From the standpoint of IS, use of RL means that not all preconditions need be hand-crafted. Preconditions that capture how one action might be more beneficial than another can be difficult to determine for dialogue designers. For example, knowing whether to first ask the number of stops or the airline, depends on the characteristics of the flights in the database, and on users' relative flexibility with these two parameters. The same problems occur for knowing under which situations to requery the database or ask for another parameter. RL solves this issue as it can explore the space of different policies to arrive at one that minimizes a dialogue cost function.

References

J. Allen and C. Perrault. 1980. Analyzing intention in utterances. *Artificial Intelligence*, 15:143–178.

P. Bohlin, R. Cooper, E. Engdahl, and S. Larsson. 1999. Information states and dialogue move engines. In *Proceedings of the IJCAI Workshop: Knowledge and Reasoning in Practical Dialogue Systems*, pg. 25–31.

P. Cohen and C. Perrault. 1979. Elements of a plan-based theory of speech acts. *Cognitive Science*, 3(3):177–212.

M. English and P. Heeman. 2005. Learning mixed initiative dialog strategies by using reinforcement learning on both conversants. In *HLT and EMNLP*, pages 1011–1018, Vancouver Canada, October.

K. Georgila, J. Henderson, and O. Lemon. 2005. Learning user simulations for information state update dialogue systems. In *Eurospeech*, Lisbon Portugal.

S. Larsson and D. Traum. 2000. Information state and dialogue management in the TRINDI dialogue move engine toolkit. *Natural Language Engineering*, 6:323–340.

E. Levin, R. Pieraccini, and W. Eckert. 2000. A stochastic model of human-machine interaction for learning dialog strategies. *IEEE Transactions on Speech and Audio Processing*, 8(1):11–23.

C. Matheson, M. Poesio, and D. Traum. 2000. Modelling grounding and discourse obligations using update rules. In *NAACL*, Seattle, May.

D. Poole, A. Mackworth, and R. Goebel. 1998. *Computational Intelligence: a logical approach*. Oxford University Press.

K. Scheffler and S. J. Young. 2002. Automatic learning of dialogue strategy using dialogue simulation and reinforcement learning. In *HLT*, pg. 12–18, San Diego.

S. Singh, D. Litman, M. Kearns, and M. Walker. 2002. Optimizing dialogue managment with reinforcement learning: Experiments with the NJfun system. *Journal of Artificial Intelligence Research*, 16:105–133.

R. Sutton and A. Barto. 1998. *Reinforcement Learning*. MIT Press, Cambridge MA.

M. Walker. 2000. An application of reinforcement learning to dialog strategy selection in a spoken dialogue system for email. *Journal of Artificial Intelligence Research*, 12:387–416.