Advances in Cognitive Systems 2 (2012) 221-238 Submitted 09/2012; published 12/2012

Online Determination of Value-Function Structure and Action-value Estimates for Reinforcement Learning in a Cognitive Architecture

John E. Laird LAIRD@UMICH.EDU
Nate Derbinsky NLDERBIN@UMICH.EDU
Miller Tinkerhess MILLER.TINKERHESS@GMAIL.COM
Computer Science and Engineering, University of Michigan, Ann Arbor, MI USA

Abstract

We describe how an agent can dynamically and incrementally determine the structure of a value function from background knowledge as a side effect of problem solving. The agent determines the value function as it performs the task, using background knowledge in novel situations to compute an expected value for decision making. That expected value becomes the initial estimate of the value function, and the features tested by the background knowledge form the structure of the value function. This approach is implemented in Soar, using its existing mechanisms, relying on its preference-based decision-making, impasse-driven subgoaling, explanation-based rule-learning (chunking), and reinforcement learning. We evaluate this approach on a multiplayer dice game in which three different types of background knowledge are used.

1. Introduction

One long-term goal of our research is to build autonomous agents that use a wide range of knowledge, including expert background knowledge and knowledge learned from experience. A challenge for such agents is how to use both background knowledge and experience to improve their behavior in novel problems with large problem spaces. One online learning mechanism that is a key learning component of many cognitive architectures, including Soar (Laird, 2012), Clarion (Sun, 2006), and ACT-R (Anderson, 2007), is reinforcement learning (RL; Sutton & Barto, 1998). In RL, an agent tunes the *value function*, which is a mapping from state-action pairs to an expected value of future reward, using its experience in an environment. In the future, the value function is then used to select actions so that the agent maximizes its expected reward. Although RL has been studied extensively, there has been little research on how it is integrated in a cognitive system, and more specifically, how the value function and its initialization are determined by an agent during execution in a novel task using background knowledge. Usually, the structure of the value function is determined by a human, with all entries initialized randomly or to a baseline value (such as 0.0). However, if our cognitive agents are to be truly autonomous, they need to have the ability to define and initialize their value functions on their own.

In this paper, we present an approach in which the value function is *incrementally* determined and *initialized* as the agent uses background knowledge to make decisions in novel situations. Once the value function's structure is determined and initialized for a given situation, the agent

© 2012 Cognitive Systems Foundation. All rights reserved.

uses reinforcement learning to dynamically update and improve the value function. The potential advantage of using background knowledge to initialize the value function is that initial performance is not random, but instead can be near expert level. The potential advantage of using background knowledge to determine the structure of the value function is that irrelevant features of the environment are ignored and relevant features are included, potentially leading to faster learning and higher asymptotic performance. The potential advantage of converting the background knowledge into a value function is that reinforcement learning can improve upon the original background knowledge, incorporating statistical regularities that it did not include.

This work extends the original Universal Weak Method (UWM; Laird & Newell, 1983), where the structure of a cognitive architecture makes it possible for available background knowledge to determine the weak method used in problem solving. Here we build on the problem-space formulation of behavior, the preference-based decision making, and the impasse-driven subgoaling of the Soar cognitive architecture that supported the original UWM, and add chunking and reinforcement learning, which together lead to the learning and decision-making methods described here (Laird, 2012). In large part, this paper is a recognition, explication, and demonstration of emergent behavior that is implicit in those mechanisms and their interaction.

Although the steps in the approach we described above are relatively straightforward, there are challenges in building a task-independent approach that can be used by a persistent agent for wide varieties of background knowledge:

C1. **Knowledge General.** The approach should work with different types of background knowledge.

C2. **Task General.** The approach should work with any task in which the agent has appropriate background knowledge and reward.

C3. **Minimally Disruptive.** The approach should not disrupt agent task processing nor require restructuring of agent processing or knowledge structures.

One ramification of challenges C1 and C2 is that this approach should not be restricted to propositional representations – it should support relational representations of task knowledge and environmental features.

Previous approaches have examined learning using specific types of background knowledge including pre-existing action models (Shapiro, Langley, & Shachter, 2001), hand-crafted state features (Korf & Taylor, 1996; Ng, Harada, & Russell, 1999), initial value functions (Maire & Bulitko, 2005; Von Hessling & Goel, 2005), as well as state-feature/structure utility advice from other knowledge sources (Shapiro, Langley, & Shachter, 2001; Dixon, Malak, & Khosla, 2000; Moreno et al., 2004). Shavlik and Towell (1989) describe KBANN, which uses a specific type of symbolic background knowledge to determine the structure and initialize the weights of a neural network (but do not apply it to reinforcement learning). One system that incorporates both analytic and inductive mechanisms in a similar manner to what we propose is EBNN (Mitchell & Thrun, 1993), which uses explanation-based learning to convert symbolic action models into neural networks, which are used as value functions for reinforcement learning. The EBNN algorithm is restricted to using action models as background knowledge and it is not incorporated into a general cognitive architecture, so it can be disruptive to how agent knowledge is formulated and used. Moreover, the analytic component does not determine the structure of the value function. Mitchell and Thrun (1996) speculate as to how a scheme such as EBNN could be incorporated into Soar, and one view of this research is as a long-delayed realization of that

vision (although our implementation differs significantly from what they propose). ACT-R (Anderson, 2007) and Clarion (Sun, 2006) both incorporate rule learning and rule tuning components; however, it is unclear how various types of background knowledge (such as action models) can be incorporated into those architectures without requiring significant restructuring of the agent processing and knowledge structures.

In the rest of this paper, we report our research in this area. In Section 2, we describe Soar and how reinforcement learning is integrated in it. Section 3 describes our approach in detail, mapping it onto the components of Soar. Section 4 describes examples of the different types of background knowledge that can be used for initializing and determining the value function. In Section 5, we provide a detailed evaluation of the approach for a multiplayer game (Liar's Dice). In Section 6, we review the challenges and how our approach meets them. We also relate our approach to other cognitive architectures and conclude with a discussion of future research.

2. Reinforcement Learning in Soar

In this section, we describe reinforcement learning and then map its components onto the corresponding representations, memories, and processes in Soar, focusing on Soar's representation of the value function. In Section 3, we build on this foundation and describe how background knowledge combines with Soar's impasse-driven subgoaling and chunking to incrementally determine and initialize the value function.

In reinforcement learning, an agent is in a state and selects between alternative actions, trying to maximize it expected future reward. The value function is a mapping from states and actions to the expected reward. Thus, to make a decision, the agent uses the value function to determine the expected value for every possible action in the current state. An action is then probabilistically selected to balance the benefit of exploitation versus exploration. Learning occurs by updating the value function for a state-action pair based on the reward received from executing the action, combined with the discounted expectation of future reward. For temporal-difference (TD) learning, this expectation is derived from the actions available in the next state. For example, the on-policy SARSA algorithm (Rummery & Niranjan, 1994) utilizes the value of the next selected action, whereas the off-policy Q-learning algorithm (Watkins, 1989) uses the greatest value of the next available actions. The value function is the locus of task-dependent knowledge. In its simplest formulation, the value function is tabular – a list with an expected value for each state/action pair.

In 2005, Soar was extended to include RL, building on the existing formulation of tasks as search through problem spaces (Nason & Laird, 2005). Soar has a working memory that is hierarchically organized in terms of states, and its processing cycle is to propose, evaluate, select, and apply *operators* that make changes to the state. These changes can initiate motor actions in an external environment, but they can also be internal, such as initiating retrievals from Soar's long-term declarative memories (semantic and episodic). The knowledge to propose, evaluate, and apply operators is encoded as production rules in Soar's long-term procedural memory. When the conditions of a rule successfully match structures in a state, it fires. The actions of a rule can propose an operator, evaluate an operator, apply an operator, or elaborate the state. All rules that match fire in parallel, so that the locus of decision making is operator selection not on determining which rule to fire. Soar supports relational representations of states and operators.

To select between operators, the operator-evaluation rules test aspects of the state and the proposed operators and create *preferences.* A fixed decision procedure interprets the preferences and selects an operator, unless the preferences are incomplete or inconsistent, in which case an impasse arises. The preferences can be either symbolic or numeric. The numeric preferences specify an operator's expected value for the state. Thus, the rules that create the numeric preferences (which we call RL-rules) provide a piece-wise (and potentially overlapping) encoding of the value function. If all aspects of the state and operators are tested and the rules are mutually exclusive, the RL-rules encode a purely tabular value function. If a common subset of the state and operators are tested by multiple rules, they provide a tile-coding. If there are multiple sets of RL-rules for different subsets of the state and operator, they encode a hierarchical tile-coding, and the preferences from multiple rules of different levels of specificity are added together to select an operator. Rules that test different aspects of the state and operator encode a coarse coding.

An operator is selected based on the preferences that have been created. The symbolic preferences filter out proposed operators, and if those preferences are sufficient to determine a single choice, then that operator is selected. Otherwise, if there are numeric preferences for the remaining operators, a probabilistic selection is made using the preference values according to a Boltzmann distribution. If some of the remaining operators do not have numeric preferences, an impasse arises and a substate is created, which leads to further problem solving (see below). After an operator is selected, rules apply it by changing the state. After application, and selection of the next operator, the numeric preferences of RL-rules for the last selected operator are updated.

3. Our Approach

In our approach, there are two different ways background knowledge can influence RL. The first is in determining the structure of the value function: which aspects of the state and operators are mapped to an expected value? If relevant features are not included, the entries in the value function will cover multiple states with different expected values, potentially making it impossible to achieve optimal performance. If irrelevant features are included, then states that should have the same expected value are treated independently, each receiving fewer updates, thereby slowing learning. Thus, selecting the right features for the value function is critical to both the speed of learning and the final asymptotic performance.

The second way background knowledge can influence RL is in initializing an entry in the value function. It is typical to initialize value functions randomly or with some baseline value. However, by initializing the value function using prior knowledge, it may be possible to greatly speed learning, reducing the need for potentially costly experience with the task.

Our claim is that the approach described below enables dynamic determination of an agent's value function, while providing initial values and meeting the challenges (C1-C3) set forth above. In our approach, an agent dynamically computes an estimate of the expected value for a novel situation using deliberate reasoning and background knowledge. The deliberation is compiled using an explanation-based generalization technique (chunking in Soar), which produces a mapping from a subset of the current state and action to the computed value, simultaneously defining and initializing a component of the value function. The value function replaces future deliberate calculations for the same situation and is then tuned by RL. In more detail:

1. The first time the agent encounters a state, it does not have a value function for evaluating each state/operator pair, leading Soar to detect an impasse in decision making.
2. In response to the impasse, Soar creates a substate in which the goal is to generate preferences so that an operator can be selected. This involves computing expected values for the proposed operators for which there is not an existing numeric preference (and thus, not a corresponding RL-rule), or performing some other analysis that results in the creation of sufficient symbolic preferences.
3. In the substate, a general task-independent operator (called *evaluate-operator*) is proposed and selected for each task operator. The purpose of evaluate-operator is to compute the expected value of each competing task operator. Once evaluate-operator is selected for a task operator, another impasse arises because there is no rule knowledge to apply it. In the ensuing substate, background knowledge is used to evaluate the task operator and compute an initial expected value for it. The types of background knowledge that can be used are quite broad, and discussed below.
4. Once the background knowledge has created an expected value, a numeric preference is created for the task operator being evaluated. This is a critical step because it converts a declarative structure (the expected value) into a preference that influences decision making.
5. As a side effect of the substate computation that generates the numeric preference, a new rule is automatically created by chunking. Chunking is a form of explanation-based learning (EBL; DeJong & Mooney, 1986), and creates rules in exactly the same format as all other rules in Soar. The action of the new rule is the numeric preference produced in the substate, while the conditions are determined by a causal analysis of the productions that fired along the path to creating the preference. The aspects of the task state and operator that were required to produce that preference become the conditions. Thus, the new rule (which is an RL-rule) tests those aspects of the task state and operator that were relevant for computing the estimated expected value and the action is the numeric preference.
6. When a similar situation (as determined by the conditions of the new rule) is encountered in the future, the rule will fire, create a numeric preference for a proposed operator, and eliminate the need to perform the deliberate reasoning. The rule is then updated by RL when the proposed operator is selected and applied. Thus, as a learning mechanism, chunking leads to more than just knowledge compilation, because it creates a representation that another learning mechanism can modify. By creating a direct mapping from the state/operator to the expected value, it produces a representation that is amenable to being updated by RL, so that the statistical regularities of the task can be captured.

4. Types of Background Knowledge

In our approach, background knowledge is used to deliberately evaluate proposed operators in substates, generating an initial estimate of the expected value, as well as indirectly determining which features of the task state and operator map the value function to the expected value. Given the generality of the problem space paradigm on which Soar is modeled (Newell, 1991); there are few limits on the types of knowledge that can be used for this purpose. The important restrictions are that the knowledge must be in some way dependent on the current situation and it must create some value that can be used as an estimate of expected future reward. Our approach by itself does not make any guarantees as to the quality of the agent's behavior. It is a conduit for converting

background knowledge into a form that can control behavior and be tuned by experience. If the estimated expected values are poor and based on irrelevant features, we expect poor performance and learning. If the expected value are accurate and based on relevant features we expect good performance and learning.

Below we provide examples of types of background knowledge that can be used with our approach. Soar agents have been developed that make use of all of these with our approach, with the exception of item 4.

1. *Heuristic Evaluation Functions:* A heuristic evaluation function is a mapping from a state to an estimate of expected value. Combined with an action model (see below) it allows the agent to predict the future reward that an operator could achieve. In Soar, the evaluation function can be encoded in rules or in semantic memory as a mapping between state descriptions and expected values.
2. *Action Models:* An action model allows the agent to predict the results of external actions on an internal representation of the task, thus allowing the agent to perform an internal look-ahead search, explicitly predicting future states. A heuristic evaluation function can be used to predict the expected reward that operator will achieve, or the agent can search until it reaches terminal states, such as in a game-playing program where it could detect win, lose, or draw. Soar supports a wide variety of approaches for encoding action models, including rules, semantic memory, episodic memory, and mental imagery (Laird, Xu, & Wintermute, 2010).
3. *Reasoning:* In some cases, an estimate of the expected value can be computed directly using features of the state and proposed operator without explicitly modeling operator actions.
4. *Analogy:* The agent can use the current state and proposed operators to retrieve a memory of a similar situation and then use historical information of the expected value of that situation as an estimate of the current situation.

With our approach, we have developed three agents, two that use a combination of action models and heuristic evaluation, and one that uses reasoning and is described in detail in the next section. The first two use one-step look-ahead hill climbing. One of these finds paths through graphs embedded in 2D space. It uses the reciprocal of Euclidian distance as an estimate of expected value. The second agent parses English sentences. It estimates the expected value of a parsing operator by applying the operator, and then retrieving from semantic memory the reciprocal of the estimated distance required to achieve a successful parse. The distance metric is computed based on an offline analysis of the Brown corpus. As expected, in both cases, the initial search depends on the quality of the distance estimates, and performance improves as reinforcement learning refines the initial approximate expected values.

5. The Dice Game

To provide an in-depth empirical demonstration and evaluation of this approach, we use a multiplayer dice game. The dice game goes by many names, including Perudo, Dudo, and Liar's Dice. The game has sufficient structure such that multiple types of background knowledge are easy to identify and useful for improving performance. Moreover, there is sufficient complexity such that the obvious types of background knowledge are incomplete, making further learning useful. This work extends previous research that focused on the integration of probabilistic background knowledge into a cognitive architecture, and only briefly touched on learning (Laird, Derbinsky, & Tinkerhess, 2011).

A game begins with the players sitting around a table, with each player having five dice and a cup. Play consists of multiple rounds. At the beginning of each round, all players roll their dice, hiding them under their cup. Players can view their own dice, but not the dice of others. The first player of a round is chosen at random. After a player's turn, play continues to the next player.

During a player's turn, an action must be taken, with the two most important types of action being bids and challenges. A bid is a claim that there is at least the specified number of dice of a specific face in play, such as six 4's. Following the first bid, a player's bid must increase the previous bid. If the dice face does not increase, the number of dice must increase. Thus, legal bids following six 4's include six 5's, six 6's, seven 2's, and so on. If a player challenges the most recent bid, all dice are revealed and counted. If the number of dice of the bid face equals or exceeds the bid, the challenger loses a die. Otherwise, the player who made the bid loses a die. A player who loses all dice is out of the game. The last remaining player is the winner.

There are additional rules that enrich the game. A die with a face of 1 is wild, and contributes to making any bid. Given the special status of 1's, all 1 bids are higher than twice the number of other bids. For example, three 1's is higher than six 6's and the next bid after three 1's is seven 2's. When a player makes a bid, they can "push" out any proper subset of their dice (usually 1's and those with the same face as the bid), exposing them to all players, and reroll the remaining dice. A push and reroll increases the likelihood of a bid being successful, and provides information to other players that might dissuade them from challenging a bid. A player can also bid "exact" once per game. An exact bid succeeds if the number of dice claimed by the most recent bid is accurate, otherwise the bid fails. If the exact bid succeeds, the player gets back a lost die; otherwise the player loses a die. Finally, a player with more than one die can "pass," which is a claim that all of the player's dice have the same face. A pass can be challenged, as can the bid before a pass. A player can pass only once with a given roll of dice.

5.1 Dice Game Agents

Our agents play using a game server that simulates random dice rolls, enforces the rules of the game, advances play to the next player, provides data on the game state, and randomly selects the first player. The agents can play against either human or other Soar agents; however, all the experiments below use only Soar agents. When it is the agent's turn, it receives information about the game that is equivalent to the information that is available to human players. This includes the number of dice under each player's cup, players' exposed dice, the dice under the agent's cup, and the history of bids. The agents are implemented in Soar 9.3.2 using approximately 340 hand-written rules. 79 of those rules are "default" rules that are used in many Soar agents and provide a general method for handling tie impasses and deliberate operator evaluation. The other rules encode task-specific knowledge about playing the game and performing the calculations described in the following paragraphs. The only additional source of knowledge is a function for calculating the probability of a configuration of dice, such as the probability that when there are eight dice, four of them have a specific face. Everything else is encoded in rules.

When it is an agent's turn, the agent first computes the number of dice of each face that it knows (those exposed from pushes plus those under its cup) and the number of unknown dice. It then determines a *base* bid. If the agent is first to bid or if the previous player made a low bid, the count for the base bid is one less than the expected number of faces given the total number of dice in play, and the face is 2. Otherwise the last bid is used as the base bid.

The agent then proposes operators for all bids that are up to one number higher than the base bid. Thus, if the base bid is six 4's, the agent proposes six 5's, six 6's, three 1's, seven 2's, seven 3's, and seven 4's. If there are dice under its cup that have the same face as the bid or are 1's, the agent also proposes bids with pushes for those dice. The agent also proposes all legal challenge, pass, and exact actions. These bids and actions are the task operators in this domain and we refer to them as *dice-game-action* operators.

The agent then selects a dice-game-action operator and sends the action to the game server. If the action is a challenge or exact, the game server determines whether it is successful, updates the number of dice for the players as appropriate, and provides feedback. The game server also provides feedback when the agent is challenged.

To select between the dice-game-action operators, the agent uses the approach described earlier. If there are sufficient preferences to select an operator, one is selected. Otherwise, a tie impasse results and those operators without numeric preferences are evaluated, which eventually leads to the creation of preferences and the selection of an operator.

To deliberately evaluate a bid, the agent can use three different types of background knowledge. The most obvious type is knowledge about the probabilities of unknown dice. Those probabilities are used to compute the likelihood that each dice-game-action will be successful. For challenges and exact bids, this is the probability that the bid will succeed, while for other bids, it is the probability that the bid will succeed if challenged. These probabilities are later used as an estimate of the expected value of the bids. Especially in games with more than two players, the probabilities for bids that are not challenges or exacts are only rough approximations because the expected value of a bid (such as three 4's) does not exactly correspond to whether it will succeed if challenged. A bid has significant value if the next player does not challenge the bid, but instead makes some other bid that ultimately results in some player besides the agent losing a die. Thus there is some positive value in making low bids because it lowers the likelihood that the next player will challenge the bid, but there is a negative value if it is so low that the player must bid again later in the round. Striking the right balance between high and low bids appears to us to be one of the types of knowledge that is hard for an expert to extract and encode, but might be possible to learn through reinforcement learning.

The second type of knowledge is a simple *model* of the previous player's bidding strategy, which is used to infer what dice are likely to be under the player's cup given their bid. The inferred dice are used in a second round of probability calculations, but only if non-model-based probability calculations do not identify a safe bid. The model will be incorrect when the player is bluffing, and it is an empirical question as to whether this knowledge is useful.

The third type of knowledge consists of *heuristics* that attempt to capture additional structure of the game that is not included in the probability of success of individual actions. For example, if a player has a valid pass, making that bid is guaranteed to avoid losing a die when it is bid. However, it is probably best to save that pass until later when the player has no other safe bids. Similarly, it is better not to push and reroll if there is another bid without a push that is unlikely to be challenged by the next player. A push reveals information to the other players and decreases the agent's options in the future.

All of these forms of knowledge (probability, model, and heuristics) are encoded as operators that are used in the deliberate evaluation of a dice-game-action operator. Their calculations are combined to create preferences for selecting the operators. The results of the probability and model knowledge lead to the creation of numeric preferences (and RL-rules), while the heuristics

Table 1. Example RL-rules learned by chunking.

```
Rule 1: If the operator is to bid five 2's with no push and
           there are zero 1's and one 2 dice and
           there are four unknown dice
        then create an operator preference of -0.8754

Rule 2: If the operator is to bid five 2's pushing one 1 and two 2's and
           the previous bid was two 1's and
           there are five dice under my cup, with one 1 and two 2's and
           the other player has only a single unknown die
        then create an operator preference of 0.12
```

lead to the creation of symbolic preferences that explicitly prefer one operator to another (and the creation of new rules that generate those preferences in the future). The learned rules that include symbolic preferences are not updated by RL.

Once an RL-rule is learned, it is updated based on reward, and expected future reward. The agent uses the obvious reward function: +1 for winning a challenge/exact bid and -1 for losing a challenge/exact bid. To align the probabilities with the reward function, the probabilities are linearly rescaled to be from [-1.0, +1.0] instead of [0.0, +1.0] when they are created.

Table 1 shows textual descriptions of RL-rules learned by an agent. Rule 1 captures the probability calculation for an unlikely bid and is typical of many rules that test the bid, the relevant known dice, and the number of unknown dice. The bid is five 2's, with one known 2 and four unknown dice, all of which must be 1's or 2's for the bid to succeed. Rule 2 is also for the selection of an operator that bids five 2's, but it was created using background knowledge that included using a model of the previous player. This operator includes pushing three dice and rerolling two, and is proposed when the opponent has only one unknown die. Thus, the agent knows there are at least three 2's and must pick up two more from the two dice it rerolls and the one die under the opponent's cup. This outcome is unlikely as is; however, the rule's initial value was computed using the opponent model. The model led the agent to conclude that in order to make the bid of two 1's, the opponent had one 1. Thus, the agent believed it needed to get only one 2 or one 1 out of the two dice it rerolls given that the previous player bid two 1's.

One characteristic of the chunks is that they are specific to the dice face being bid and the existence of 1's. Given the special nature of 1's as wild cards, they must be accounted for. Beyond the specificity in testing the dice faces, the rules generalize over other aspects of the situation that were not tested when they are learned. For example, Rule 2 does not test which other dice are under its cup, except that they are not 1's or 2's. Similarly, Rule 1 does not test the previous bid, nor does it test which other non-1 or non-2 dice are known to the agent, as they are irrelevant to computing the bid's probability.

Because the determination of the conditions of a new rule is dependent on the knowledge used in the subgoal, we expect that the value functions learned from different initial knowledge to be different. If the model is used, the learned rules will test features of the previous bid, whereas if neither the model nor the heuristics are used, the learned rules will test only the player's dice, the number of dice under cups, and the exposed dice. When more features are tested, those rules will be more specific, and more rules will be learned. We expect these factors to influence the speed and quality of learning, as well as the number of rules that are learned.

5.2 Evaluation

To evaluate the performance of agents using learning, we used three-player games. Such games take significantly less time to play than games with more players and have significantly smaller state spaces than four-player games (~200K-2M RL-rules versus ~800K-8M RL-rules). We used Q-learning with a Boltzmann distribution for selection between operators with numeric preferences. The learning parameters were learning rate = 0.3; discount rate = 0.9; and temperature = 0.008. The first two are the default values in Soar (and verified by parameter sweeps). The temperature was determined by a parameter sweep. The agents play against the best non-learning agent, which includes all the background knowledge: probability, opponent model, and heuristics. The agents play 1,000 games of testing (learning disabled) alternated with 1,000 games of training (learning enabled), for a total of 20 trials of training and testing, resulting in 20,000 total games of training. Each trial (2,000 games) takes between 70-90 minutes real time, with each game taking 2-3 seconds. Each game takes 10 to 20 rounds of play (players can win back a die with an exact). All data are the average of four independent runs.

We evaluate four agents with different combinations of background knowledge. These include probabilities [P], probabilities and model [PM], probabilities and heuristics [PH], and all three [PMH]. Probability knowledge is included in all agents because without it there are situations in which the agents cannot make decisions. The need for probability knowledge is because the other forms of knowledge do not provide a policy for all states and operators – they only cover a subset of the space so that there are many situations in which the heuristic and model do not apply. This is a feature of the task and types of knowledge and is independent of are approach or our implementation in Soar. The opponent agents are the PMH agent but without learning. Thus, given that there are three players, a non-learning PMH agent is expected to win 333 (33%) of the games because it is playing against two copies of itself.

To verify that the background knowledge is useful, we played an agent that randomly selected from among the legal operators against two PMH agents. The random agent won 0.6% of the games. For more analysis of the value of the different types of background knowledge in non-learning experiments, see Laird, Derbinsky, and Tinkerhess (2011).

The first question we consider is whether this approach to learning leads to improvements in performance. Figure 1 shows the performance of the four agents over 20 trials. As expected, the PMH agent starts out best, winning ~33% of its games, while the other agents, with less background knowledge, do not perform as well. However, they all significantly improve performance in the first training trial so that in the following testing trial they are all performing better than the non-learning PMH agent. These results substantiate the claim that RL can improve on the original background knowledge for this task. This is a function of the fact that our background knowledge is only approximately correct. For a task in which the background knowledge correctly estimates the expected value of every operator in every situation, we expect no improvement.

The second question we consider is what effect different types of background knowledge have on learning. Background knowledge can influence learning in three different ways:

1. It provides initial estimates of expected values;
2. It eliminates poor choices from consideration; and
3. It determines the structure of the value function based on what aspects of the state are tested.

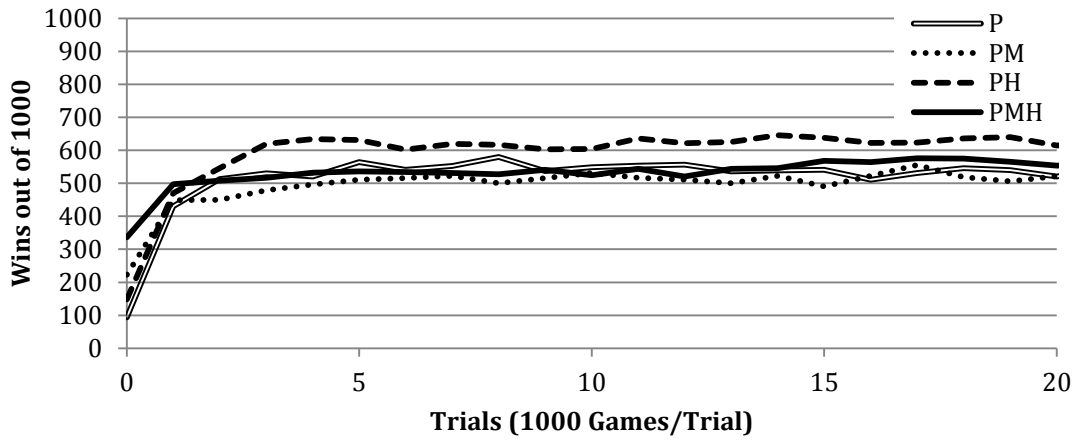

Figure 1. Learning curves for agents with learned rules initialized by internal deliberation in subgoals.

We consider each of these in turn, and we return later to other features of Figure 1, such as that PH learns faster and has better performance than the other agents.

To evaluate whether the initial estimates created by the background knowledge are useful, we created variants of the agents (labeled with -0), where the probability calculation is still performed (so that the same aspects of the state are tested), but the computed estimated expected value is 0.0. These values are then tuned by RL, but since these agents do not use the probability calculations as initial estimates of expected value, we can discover if those estimates are useful by comparing their performance to the agents in Figure 1. For these agents, a temperature of .08 was used, which led to faster learning by encouraging more exploration.

Figure 2 shows that the agents whose expected values were initially set to 0.0 perform worse

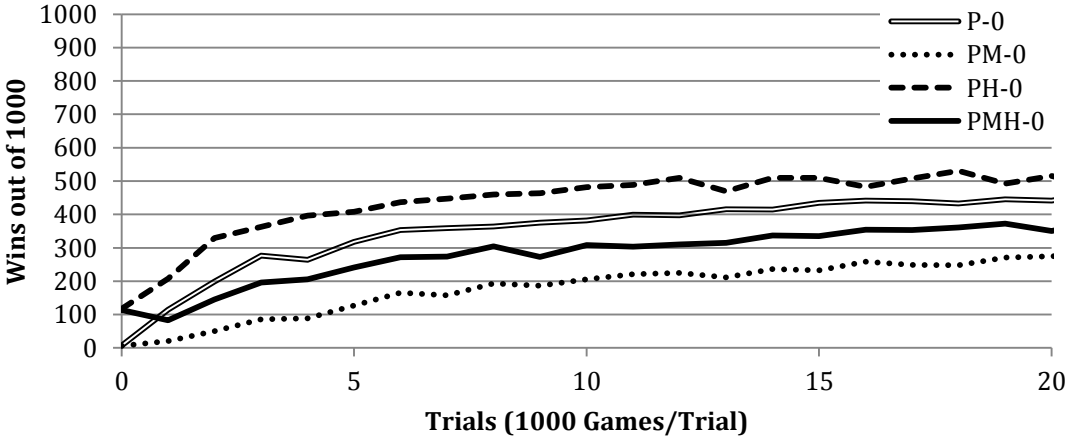

Figure 2. Learning curves for agents with learned rules initialized to zero.

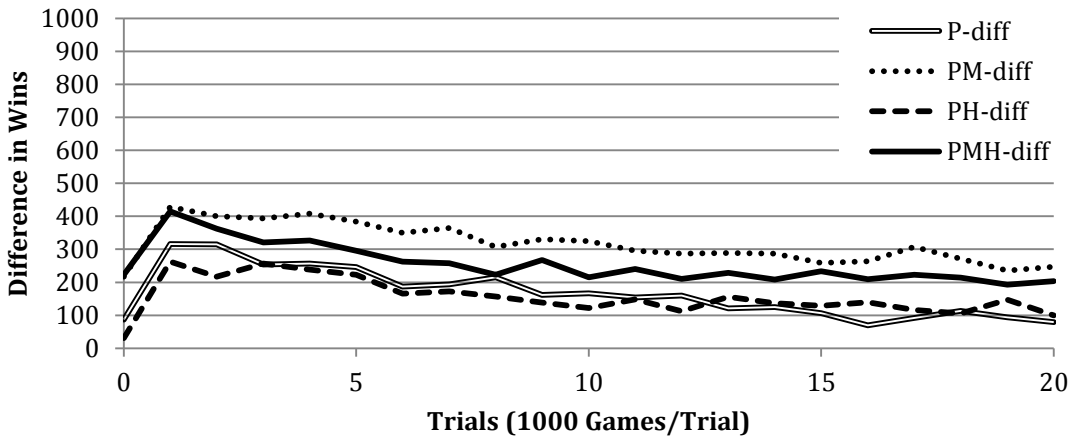

Figure 3. Differences in agent performance with and without initial value estimates.

than the agents using the probability calculations (Figure 1), verifying our claim. The agents in Figure 2 show only gradual learning, whereas the agents in Figure 1 initially learn quickly, as the initial expected values focus exploration on the best actions. Figure 3 shows the difference in performance between the corresponding agents in Figure 1 and 2, so that a higher value in the figure means better performance with initialization. As expected, the initialization has its biggest impact during the first trial. The difference lessens over time, but even after 20,000 games, there is still a significant advantage to starting with good estimates of the expected values.

We next consider how the background knowledge eliminates poor choices from consideration. By eliminating poor choices, an agent will not waste time exploring operators that are obviously bad, which will focus both performance and learning on the best operators. This is the main role of heuristics and to a lesser extent the initial value estimates, leading us to expect that the agents with heuristics will outperform those without. This is confirmed in Figures 1 and 2, where PH dominates P and to a lesser extent, PMH dominates PM, suggesting that in this domain, the heuristics help the agents avoid bad choices. It also suggests that the knowledge in the heuristics is difficult to learn using RL, at least given the number of trials in our experiments. Thus, our hypothesis is that the heuristics we implemented test combinations of features that are not captured in the other types of background knowledge we evaluated. For example, the heuristics for avoiding an operator that rerolls dice test that there is a safe bid so that there is no need to reroll. The rules that avoid using a valid pass include similar conditions. This type of knowledge is not included in the learned RL-rules because they only include conditions that are specific to the value of a single bid, independent of all of the other bids. Thus, they are unable to distinguish between situations in which there is a safe bid (so that the push or pass bid should be avoided) and those in which there is no safe bid (so that the bid or pass bid should be used).

To further explore this hypothesis, we did a small study in which we disabled the learning of chunks in the PH agent for the heuristic-based evaluations. For this agent, heuristic reasoning is used initially, but it is eventually replaced by RL-tuned knowledge (whereas in the original PH agent, rules are learned that include the heuristic knowledge that continues to be used through all trials). As shown in Figure 4, the modified agent initially learns as quickly as the PH agent does

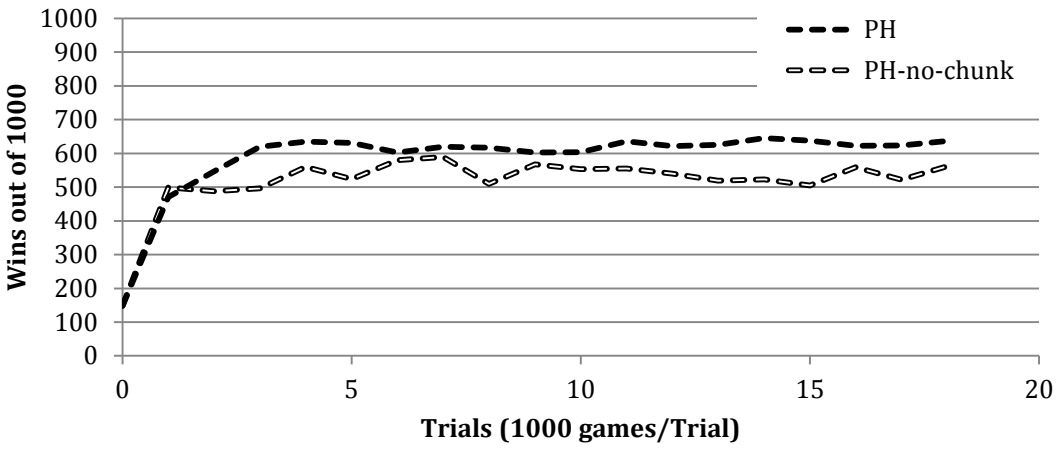

Figure 4. Performance of PH agent with and without chunking of H.

because the heuristics are initially being used to focus the exploration away for poor choices. However, as the heuristics are replaced by the purely RL-tuned knowledge, the RL knowledge is unable to capture the co-occurrence of features encoded in the heuristics. This in part provides some insight into Soar's policy of considering numeric preferences only for those operators that are not filtered out by symbolic preferences. The symbolic preferences limit the branching factor so that reinforcement learning need only learn to distinguish between "good" choices. Although these results are not directly related to the core claims of the paper, they demonstrate that there can be positive interactions between knowledge that is tuned by RL and other knowledge (the heuristics) that is not tuned by RL.

The final way in which background knowledge in our agents can influence learning is by determining the features that are tested by the value function. Including the heuristics (H) does not change the structure of the value function (although it leads to more rules being learned), so that the P and PH agents have the same value function structure for updating by RL (but get different experiences because of the effect of the heuristics), as do the PM and PMH agents. The PM and PMH agents test more features, specifically some related to the bid of the previous player, which greatly increase the size of the state space and the number of RL-rules they learn. As originally shown in Figures 1 and 2, the types of background knowledge influence the speed of learning. In this task, adding the tests related to the bid of the previous player slows learning. Our hypothesis is that this is because there are many more rules learned when those features are included so that the PM and PMH agents do not get as many updates from RL for their rules. The results suggest that the extra advantages of having a fine-grained value functions are outweighed by the lack of updates. This could be because the finer-grained value functions really do not capture any important distinctions so that the PM and PMH agents are over fitting the problem, or it could be because the state space is so large that the PM and PMH agents are not getting sufficient updates.

We first verify that the PM and PMH agents learn significantly more RL rules than the non-M agents do. Figure 5 shows the number of rules learned for the agents from Figure 1

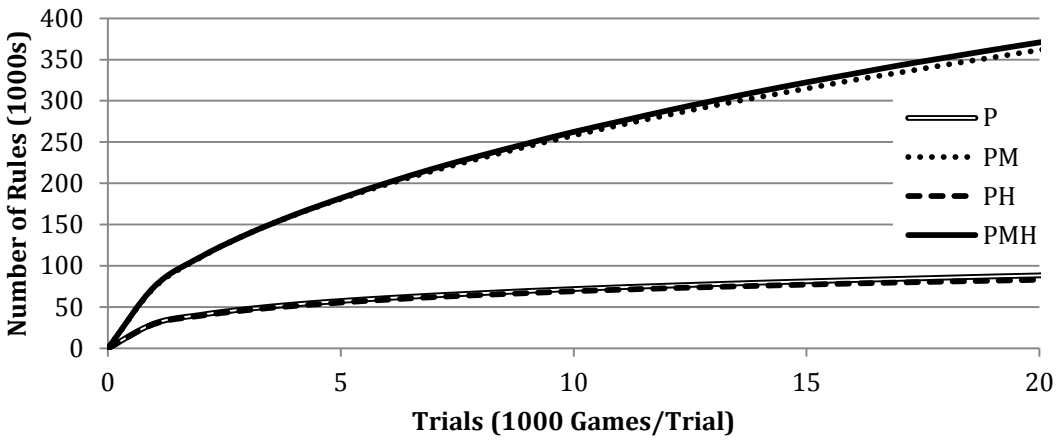

Figure 5. Number of RL rules learned for different agent types.

(approximately an equal number of additional non-RL rules are learned for the PH and PMH agents). As expected, the P and PH agents learn the same number of RL rules, as do PM and PMH agents, but many more than P and PH. Moreover, the PM and PMH agents are continuing to add more rules at a high rate, giving them the potential of continuing to improve and possibly achieving better long-term performance, although they would probably require hundreds of thousands of more trials.

To further explore these agents, we examine the number of times the RL-rules are updated. One might expect that all rules are equally likely to be updated, because any part of the state space is equally likely to be visited. This is not true for two reasons. First, once the agent knows that there are actions to avoid, it rarely if ever selects those actions. Both the heuristics and the initial expected value estimate provide strong biases away from many actions, restricting which ones are updated. Second, every game *begins* with every player having five dice, where there are a huge number of possible states, but every game *ends* with two players facing off with at least one player having a single die, so that at that point, there are only a limited number of states. Thus, over the life of the agent, it will experience states with only two players many more times than other states, and in terms of states with two players, it will also experience the states where each player has only a few dice, both because there are many fewer of them, and because every game must end with one player have no dice.

The combination of these task characteristics lead to an extremely skewed distribution of updates across rules as illustrated in Figure 6. The data for this figure are from a short run of the P agent, but they are representative of the data from other agents. The figure shows the number of updates per rule sorted by number of updates from left to right. The y-axis is logarithmic and shows the extreme distribution of updates, which is found in all agents. The total number of rules in this example is about 30,000 with the majority of them having no updates. (The rules with no updates are not displayed in the graph given the logarithmic scale.) An examination of the learned rules confirms that the rules with the most updates are for the end game, when there are two agents and one has a single die. These data help explain the fast initial learning in Figure 1 – there are a few situations that occur frequently, so learning how to respond in them can significantly

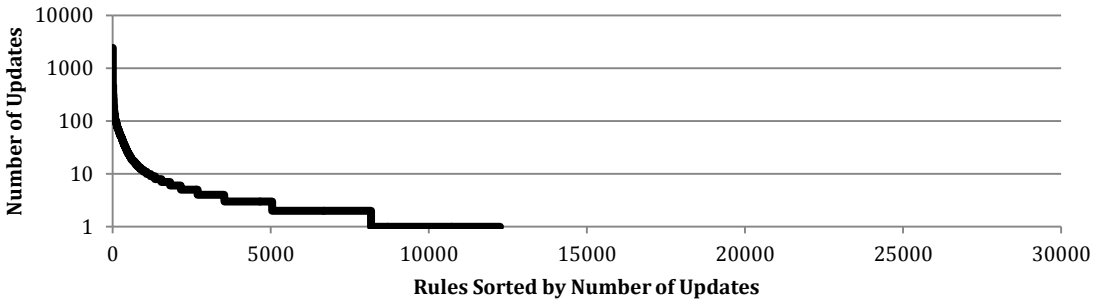

Figure 6. Number of updates for rules during run of the P agent.

improve performance; however, beyond those situations, the state space grows rapidly so that even though there can be continued improvement for all agents (including P and PH), it will be very slow. This also explains why the more detailed state space available in the PM and PMH agents does not significantly help them in our experiments. They do improve initially, but given their huge state spaces their improvement is very slow because they are rarely returning to a state they have experienced in the past.

6. Discussion

The agents in the previous section demonstrate that our approach is effective at incrementally determining both the value-function structure and initial values for reinforcement learning. The agents demonstrate that computing the estimate for the expected value using background knowledge is useful in the dice domain and that knowledge is converted into a form that is then usefully updated by RL through experience. The effectiveness of using background knowledge to determine the structure of the value function is less clear. All agents do learn quickly with the defined value functions, but there are clearly tradeoffs between the specificity of the value function and the speed of learning. We will return to this issue later.

In terms of the original challenges, our approach meets all of them. The approach is knowledge general in that it can use many different forms of background knowledge as described in Section 4, with multiple forms of background knowledge demonstrated in Section 5. That section also demonstrated that the approach is flexible enough to incorporate heuristic knowledge that is not directly related to expected reward, but which can combine with such knowledge, as in the PH agent. It is task general. Although we have focused on one task, this approach has been used on other tasks with different types of background knowledge. It can be used in any task in which there is background knowledge that produces an estimate of future reward, although its effectiveness depends on the quality of that background knowledge. Finally, the approach does not disrupt agent processing nor require any redesign of the agent's original knowledge, at least within the confines of how agents are developed within the problem-space computational model of the Soar architecture. The agents used in Section 5 were originally developed as non-learning agents before we conceived of the learning approach described in this paper. With only minimal changes and the enabling of chunking and reinforcement learning, they were used to generate the data as reported in the figures.

This approach did not have any impact on the reactivity of the agents. The inclusion of chunking, reinforcement learning, or the addition of hundreds of thousands of rules did not significantly increase Soar's cycle time (which is < 1 msec.). In contrast to other architectures, where additional knowledge can slow execution and lead to a utility problem (Minton, 1990), Soar's rule matching scales well, even to millions of rules (Doorenbos, 1994). There is a memory cost associated with each new rule (about 1K), and we have developed a forgetting mechanism that uses base-level activation to identify rules that can be forgotten, thereby significantly decreasing the memory footprint of our agents (Derbinsky & Laird, in press).

Although we evaluated our agent in three-player games, our agent can play in games with any number of players because there is no fixed value function that must be expanded as more players are included. The reasoning for making and evaluating bids uses the current number of players and their available dice, how many ever there are. However, as in the current agent, the value of reinforcement learning will be minimal when there are large numbers of dice in play because the state space is so large, so that the agent's performance will be determined by the probability calculations, the heuristics, and the agent model.

The structure of the Soar architecture was instrumental in realizing this approach. The key components are: production rules for preference-based decision making, which support relational representations of the value function and a distributed, non-tabular value function; impasse-driven subgoaling, so that background knowledge is used only when RL-rules have not yet been learned for a situation; the ability to use deliberately computed expected values for operator selection; chunking which dynamically compiles the use of background knowledge into the value function; and reinforcement learning, which updates the RL-rules.

Our approach could potentially be used in other architectures, but they must support the capabilities described above. For example, they must have a way of deliberately evaluating bids using background knowledge and then converting a declarative representation of the value of the task operator into a form that can be updated by RL. Clarion does have two levels of reasoning: a symbolic level with declarative structures and a lower level neural network. ACT-R has a production compilation mechanism that can convert multiple rule firings that process declarative structures into a single rule; however, its representation of the expected value is the rule's activation. Our approach requires the conversion of a declarative structure (e.g., the probability estimate in the dice game) into the expected value used for decision making. This would involve converting the declarative value from the symbolic processing in Clarion into a form where it initializes neural connections. In ACT-R, this would involve converting the declarative value represented in working memory into a form where it initializes the activation of a new rule. Based on our understanding, neither of these conversions is currently possible in these architectures and allowing them would require significant changes.

This work also raises issues for future research. First, incorporating more background knowledge can lead to slower learning because it can define a much larger space for learning. This was demonstrated in our experiment by the addition of the opponent model, which improved initial performance, but by testing additional features, it led to a significant increase in the specificity of the rules that represented the value function. The more accurate initial probability estimates were offset by slower learning. More broadly, in this approach, an agent is a prisoner to the distinctions made by its background knowledge. At one extreme, if the background knowledge incorporates too many features, learning via RL is slowed. At the other extreme, if the background knowledge makes insufficient distinctions, performance and learning will be poor.

Because of these issues, we are actively investigating methods that start with general background knowledge, and then incrementally incorporate more detailed knowledge for those decisions where the initially learned value function is not sufficiently discriminating. For example, an agent might initially use its background knowledge to compute probabilities but ignore the previous bid. This will lead to a smaller value function and fast initial learning. However, for those decisions where the value function does not converge (the magnitude of updates are significantly above average), additional background knowledge (such as the model of the previous player) could be used to learn more a specific value function. With this approach, the agent could deliberately reason about what features should be added to the value function, thus performing automated feature engineering. This would lead to a non-uniform hierarchical tile-coding, and require changes to how impasses are determined in Soar. This research draws on prior work on non-relational hierarchical tile coding (Munos & Moore, 1999; Reynolds, 2000; Whiteson, Taylor, & Stone, 2007).

Another line of research is to explore how the agent can learn (possibly with completely different techniques) the types of background knowledge provided to our agents, which in the dice game included symbolic heuristics not easily captured by reinforcement learning, the knowledge that probability calculations would be useful for this task, and the opponent model knowledge.

Acknowledgments

This research was supported by contract No. FA2386-10-1-4127, funded by the Air Force Office of Scientific Research.

References

Anderson, J. R. (2007). *How can the human mind exist in the physical universe?* New York. Oxford University Press.

Derbinsky, N., & Laird, J. E. (in press). Effective and efficient forgetting of learned knowledge in Soar's working and procedural memories. *Cognitive Systems Research*.

DeJong, G. F., & Mooney, R. J. (1986). Explanation-based learning: An alternative view. *Machine Learning, 1*, 145-176.

Dixon, K. R., Malak, R. J., & Khosla, P. K. (2000). *Incorporating prior knowledge and previously learned information into reinforcement learning* (Technical Report). Institute for Complex Engineered Systems, Carnegie Mellon University, Pittsburgh, PA.

Doorenbos, B. (1994). Combining left and right unlinking for matching a large number of rules. *Proceedings of the Twelfth National Conference on Artificial Intelligence* (pp. 451-458).

Korf, R. E., & Taylor, L. A. (1996). Finding optimal solutions to the twenty-four puzzle. *Proceedings of the Thirteenth National Conference on Artificial Intelligence* (pp. 1202-1207). Portland, OR: AAAI Press.

Laird, J. E. (2012). *The Soar cognitive architecture*. Cambridge, MA: MIT Press.

Laird, J. E., Derbinsky, & N. Tinkerhess, M. (2011). A case study in integrating probabilistic decision making and learning in a symbolic cognitive architecture. *Papers from the 2011 AAAI Fall Symposium Series: Advances in Cognitive Systems*. Arlington, VA: AAAI Press.

Laird, J. E., & Newell, A. (1983). A universal weak method: Summary of results. *Proceedings of the International Joint Conference on Artificial Intelligence* (pp. 771-773). Los Altos, CA: Kaufman.

Laird, J. E., Xu, J. Z., & Wintermute, S. (2010). Using diverse cognitive mechanisms for action modeling. *Proceedings of the Tenth International Conference on Cognitive Modeling*. Philadelphia, PA.

Maire, F., & Bulitko, V. (2005). Apprenticeship learning for initial value functions in reinforcement learning. *Proceedings of the IJCAI 2005 Workshop on Planning and Learning in a priori Unknown or Dynamic Domains* (pp. 23-28). Edinburgh, Scotland.

Minton, S. (1990). Quantitative results concerning the utility problem in explanation-based learning. *Artificial Intelligence, 42*, 363–392.

Mitchell, T. M., & Thrun, S. (1993). Explanation-based neural network learning for robot control. In S. J. Hanson, J. Cowan, & C. L. Giles, (Eds.), *Advances in Neural Information Processing Systems 5*, 287–294, San Mateo; CA, Morgan Kaufmann.

Mitchell, T. M.,& Thrun, S. (1996). Learning analytically and inductively. In D. Steier & T.M. Mitchell (Eds.), *Mind matters: A Tribute to Allen Newell*. Hillsdale, NJ: Lawrence Erlbaum.

Moreno, D., L., Regueiro, C. V., Iglesias, R., & Barro, S. 2004. *Using prior knowledge to improve reinforcement learning in mobile robotics* (Technical Report). University of Essex, Essex, UK.

Munos, R., & Moore, A. (2002). Variable resolution discretization in optimal control. *Machine Learning, 39*, 291-323.

Nason, S., & Laird, J. (2005). Soar-RL: Integrating reinforcement learning with Soar. *Cognitive Systems Research, 6*, 51-59.

Newell, A. (1991). Reasoning, problem solving and decision processes: The problem space as a fundamental category. In N. Nickerson (Ed.), *Attention and performance VIII*, 693-718. Hillsdale, NJ: Lawrence Erlbaum.

Ng, A. Y., Harada, D., & Russell, S. J. (1999). Policy invariance under reward transformations: Theory and application to reward shaping. *Proceedings of the Sixteenth International Conference on Machine Learning* (pp. 278-287). Bled, Slovenia.

Reynolds, S. I. (2000). Decision boundary partitioning: Variable resolution model-free reinforcement learning. *Proceedings of the Seventeenth International Conference on Machine Learning*. Morgan Kaufmann.

Rummery, G. A., & Niranjan, M. (1994). *On-line Q-learning using connectionist systems* (Technical Report CUED/F-INFENG/TR 166). Engineering Department, Cambridge University, Cambridge, UK.

Shapiro, D., Langley, P., & Shachter, R. (2001). Using background knowledge to speed reinforcement learning in physical agents. *Proceedings of the Fifth International Conference on Autonomous Agents* (pp. 254-261). Quebec, Canada.

Shavlik, J. W., & Towell, G. G. (1989). An approach to combining explanation-based and neural learning algorithms. *Connection Science, 1*, 231-253.

Sun, R. (2006). The CLARION cognitive architecture: Extending cognitive modeling to social simulation. In R. Sun (Ed.), *Cognition and multi-agent interaction*. New York: Cambridge University Press.

Sutton, R., & Barto, A. G. (1998). *Reinforcement learning*. Cambridge, MA: MIT Press.

von Hessling, A. & Goel, A. (2005). Abstracting reusable cases from reinforcement learning. *Proceedings of the Sixth International Conference on Case-Based Reasoning* (pp. 227–236).

Watkins, C. J. C. H. (1989). *Learning from delayed rewards*. Doctoral Dissertation, University of Cambridge, UK.

Whiteson, S., Taylor, M. E., & Stone, P. (2007). Adaptive tile coding for value function approximation (Technical Report, AI-TR-07-339). Computer Science Department, University of Texas, Austin.