In Proceedings of the Sixth International Conference on Cognitive Modeling, 208-213.
Mahwah, NJ: Lawrence Earlbaum.

Soar-RL: Integrating Reinforcement Learning with Soar

Shelley Nason (snason@umich.edu)
University of Michigan, 1101 Beal Ave.
Ann Arbor MI 48109-2110 USA

John E. Laird (laird@umich.edu)
University of Michigan, 1101 Beal Ave.
Ann Arbor MI 48109-2110 USA

Abstract

In this paper, we describe an architectural modification to Soar that gives a Soar agent the opportunity to learn statistical information about the past success of its actions and utilize this information when selecting an operator. This mechanism serves the same purpose as production utilities in ACT-R, but the implementation is more directly tied to the standard definition of the reinforcement learning (RL) problem. The paper explains our implementation, gives a rationale for adding an RL capability to Soar, and shows results for Soar-RL agents' performance on two tasks.

Introduction

The Soar architecture has been used extensively, both for developing AI applications and cognitive models. One of its strengths has been the ability to efficiently represent and use large bodies of symbolic knowledge to solve a wide variety of problems using many different methods. It dynamically combines available knowledge for decision-making, and can dynamically create subgoals whenever the knowledge for a decision is incomplete or inconsistent. Soar can also compile the problem solving in subgoals into rules, using a process called *chunking*, so that over time, problem solving in subgoals is replaced by rule-driven decision making. Chunking has proved to be extremely versatile because it stores away whatever problem solving is performed in a subgoal, allowing Soar programs to learn using a wide variety of methods, including explanation-based learning, macro-operator learning, strategy acquisition, learning by instruction, and many others. In general, Soar's processing is symbolic, and although that is sufficient (and necessary) for a wide variety of cognitive activities, it is inadequate (or at the very least extremely inefficient) when encoding probabilities and numeric rewards.

While Soar has strengths in knowledge-rich symbolic reasoning and learning and weaknesses in knowledge-lean, statistical-based learning, the strengths and weaknesses of reinforcement learning (RL) techniques are the reverse. They are successful at capturing statistical regularities related to the expected reward that an agent will receive, but can not encode and effectively use large bodies of symbolic knowledge. In this paper we will present an initial integration of reinforcement learning with Soar, enriching the learning capabilities as well as the representation of knowledge in Soar, while at the same time developing a unique integration of reinforcement learning with symbolic, knowledge-rich reasoning. Specifically, Soar supports dynamic hierarchical task-decomposition, meta-reasoning, and the ability to enrich the state descriptions through internal abstractions. All of these capabilities both complicate and enrich reinforcement learning. This integration requires structural changes to the Soar architecture and we will refer to the unification as Soar-RL.

In the remainder of this paper we first present a simplified description of Soar and the extensions we have made to incorporate reinforcement learning. We then demonstrate the implementation on two simple tasks, highlighting the contributions RL makes to Soar, as well as the capabilities Soar-RL provides beyond standard reinforcement learning. We also compare and contrast Soar-RL to ACT-R, which incorporates a rule-tuning mechanism, comparable to reinforcement learning. We conclude with future directions.

Soar

The structure of Soar's memories is shown in Figure 1. Soar has a declarative working memory that contains its representation of the current situation using labeled graph structures, organized in a hierarchy of states/goals. All long-term procedural knowledge is encoded as production rules. Whenever a rule's conditions match working memory, the rule is fired and its actions performed. Actions may involve adding or removing structures from working memory. They may also create preferences used to select operators.

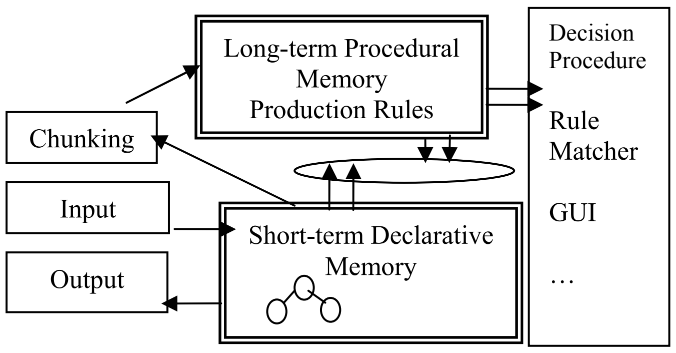

Long-term Procedural Memory Production Rules
Decision Procedure
Rule Matcher
GUI
…
Chunking
Input
Output
Short-term Declarative Memory

Figure 1: Soar's Structure

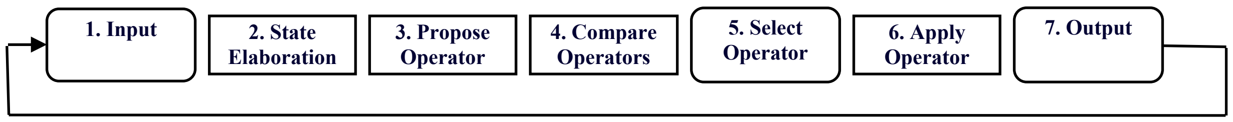


Figure 2: Soar's Decision Cycle

Soar's learning mechanism, chunking, monitors problem solving and automatically creates new rules, which are added to long-term memory during execution.

Soar's basic reasoning cycle is illustrated in Figure 2.

1. Input. Changes to perception are processed and Soar's perceptual buffer in working memory is updated.
2. State elaboration. Rules that newly match are fired in parallel to retrieve relevant information. For example, in a robotic task, a rule might test the distance to the object and the robot's available reach and determine if the object is within reach.
3. Proposing operators. Rules can propose operators by creating *acceptable* preferences for specific operators. In general, the rules' conditions test the situation so that an operator is proposed only when it is relevant.
4. Comparing and evaluating operators. Rules can test the proposed operators and other features of the situation and create preferences, which make assertions about the absolute or relative merit of the operators. Multiple preferences can be generated for a single operator.
5. Selecting the current operator. The preferences are evaluated to select the current operator. If the preferences are insufficient or contradictory, an impasse ensues and Soar creates a substate in which the goal is to resolve that impasse. This provides Soar with meta-reasoning so that it can reflect on its own processing.
6. Applying operator. Rules match against the currently selected operator and the state and make changes to the state, including creating motor commands.
7. Output. All new motor commands are processed.

Stages 2-4 are intermixed during execution. The only decision making in Soar is the selection of operators, which is the sequential bottleneck in Soar, as only a single operator can be selected for a state at a time (rules fire in parallel).

Soar's original set of symbolic preferences included preferences specifying that operators should be rejected, that one operator is better than another, or that an operator can be a default. Recently, we have added numeric preferences, which allow a real number to be associated with an operator (Wray & Laird, 2003). The numeric preferences are considered only if the symbolic preferences are insufficient for determining a single best choice. For each candidate operator O_i, the values of all numeric preferences proposed for O_i are summed into a total score $Sum(O_i)$ for the operator. The winning operator is chosen probabilistically according to the Boltzmann distribution:

$$\frac{e^{Sum(O_i)/Temperature}}{\sum_i e^{Sum(O_i)/Temperature}}.$$

Temperature is a parameter that controls the "peakedness" of the probability distribution. These selection rules provide a means of incorporating domain-specific probabilistic selection knowledge.

Adding Reinforcement Learning

The core problem in reinforcement learning is to learn how to act in the world so as to maximize a reward signal. In most approaches, an agent learns a value function, a mapping from either a state or a state-operator pair to an estimation of the expected sum of future rewards that can be achieved from that state or after applying that operator in that state. In Soar-RL, we use numeric preferences to represent a state-operator value function. We have further extended Soar by adding a reward as one of the inputs from the environment. The value of the reward can also be modified by internal knowledge (additional rules) that generates a reward for subgoal achievement.

One recurring problem in RL is how to represent the value function for a large state or state-operator space. The simplest way to store a value function is as a table with all the possible states or state-operator pairs enumerated with their associated values. However, this representation is both bulky and slow, since it does not allow learned values to be generalized over sets of states, and so is impossible for the sorts of tasks we expect to encode in Soar.

By summing over the values suggested by multiple rules, the recommendation of the numeric preferences shares a characteristic with the recommendation of the symbolic preferences – they are both the result of the combination of different pieces of advice, each of which covers a portion of the state-operator space. Each individual rule can be considered to be a specialized complex feature detector, and the predicted utilities for the operators are the runtime linear combination of the values of these complex features. Learning linear functions over features is one of the best understood methods of value function approximation in RL. Automatically constructing features, however, is not well understood, and this is part of our research (necessary, for an agent with many tasks; helped by the flexibility of Soar's language). Incorporating RL in Soar has two parts, first, adjusting the values of numeric preferences for existing rules, and later, creating new rules that test different feature sets while creating numeric preferences.

Adjusting Numeric Preferences

A numeric preference for an operator is generated by a rule associating a numeric value with a particular set of features in working memory. This set should incorporate features of the operator and the current state, possibly including aspects such as the current goal and surrounding context. Such a rule can be general enough to apply across many different states and operators if it is selective in the features it tests. As an example, a maze-world agent might have the rule[1]:

```
sp {RL*rule1
   (state <s> ^task mazeworld
              ^location C5
              ^destination A1
              ^operator <o>)
   (<o> ^name move
        ^direction east)
  -->
   (<s> ^operator <o> = -.82)}
```

Figure 3a

The RL task is to adjust the value of this rule (currently, -.82) as the agent encounters rewards in the world. The aim of these adjustments is that the summed value of numeric preferences for operator **o** given the working memory state **s** should approximate the action value, notated Q(**s**,**o**). The action value is the expected sum of rewards to be received after choosing operator **o** in state **s**, and thereafter following the agent's current policy. Such adjustments, combined with our soft greedy policy for operator selection, move the agent toward actions that maximize its expected reward.

Soar-RL performs updates of numeric preferences immediately - the preferences that fired for the operator selected on the tth decision cycle are updated during the (t+1)st decision cycle – and so manages to store very little history. The updating procedure is a variant of the SARSA algorithm (Rummery & Niranjan, 1994), which translates to the following steps in Soar-RL:

1. On cycle t, operator $\mathbf{o_t}$ is selected. This selection may or may not result solely from symbolic preferences, but, in either case, any numeric preference rules for $\mathbf{o_t}$ in state $\mathbf{s_t}$ are fired. This, possibly empty, set of instantiated numeric preference rules is stored, as is the sum over these preferences, which represents Q($\mathbf{s_t}$,$\mathbf{o_t}$).
2. The system fires rules to apply the operator $\mathbf{o_t}$, which in turn leads to new rules matching to select the following operator. At this point, the reward $\mathbf{r_t}$ for this decision phase is recorded and the next operator $\mathbf{o_{t+1}}$ is selected.
3. By this stage, the system has summed the numeric preferences for $\mathbf{o_{t+1}}$, the quantity representing Q($\mathbf{s_{t+1}}$,$\mathbf{o_{t+1}}$). The estimate Q($\mathbf{s_t}$,$\mathbf{o_t}$) will be improved by adjusting it toward $\mathbf{r_t}$ + Q($\mathbf{s_{t+1}}$,$\mathbf{o_{t+1}}$). More precisely, Q($\mathbf{s_t}$,$\mathbf{o_t}$) is updated according to:
$$Q(s_t,o_t) = Q(s_t,o_t) + \alpha(r_t + \lambda \times Q(s_{t+1},o_{t+1}) - Q(s_t,o_t))$$
where α is a learning rate and λ is the rate at which future rewards are discounted, both currently constant values set by hand.
4. The update is distributed over the saved instantiated preference rules for $\mathbf{o_t}$ by dividing the portion $\alpha(r_t + \lambda \times Q(s_{t+1},o_{t+1}) - Q(s_t,o_t))$ **(Formula 1)** by the number of preference rules and adding this quantity to the numeric value of each rule.

[1] Note on Soar syntax. The If and Then portions of a rule are separated by -->. The syntax of a condition is (identifier ^attribute value). And anything in < > brackets is a variable.

For instance, a maze-world agent chooses a **go east** operator that has a summed value –5.82 computed from the preferences in Figure 3a and Figure 3b:

```
sp {RL*rule2
   (state <s> ^task mazeworld
              ^has_monster east
              ^operator <o>)
   (<o> ^name move
        ^direction east)
  -->
   (<s> ^operator <o> = -5)}
```

Figure 3b

If the total update (**Formula 1**) computed on the next step is –1, then these two rules would be updated to have the actions (<s> ^operator <o> = -1.32) and (<s> ^operator <o> = -5.5) respectively.

Currently, the Soar-RL agent will do no reinforcement learning unless the programmer provides numeric preference rules for it to update. These rules can be written in the form of variabilized prototypes such as

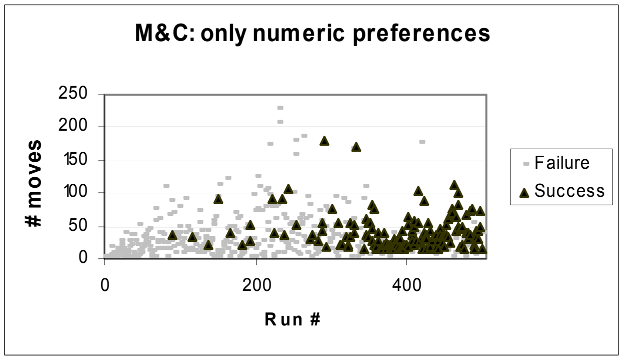
```
sp {RL*prototype
 (state <s> ^task mazeworld
            ^location <c>
            ^destination <d>
            ^operator <o>)
 (<o> ^name move
      ^direction <direction>)
-->
 (<s> ^operator <o> = 0)}
```

When this rule fires in a specific situation, it will build a new version of itself with the variables <c>, <d>, and <direction> replaced by constants, as in Figure 3a. At that point and thereafter, this new rule will have its value updated by the procedure described above. Using this prototype allows the agent to associate values with all combinations of location, destination, and direction without hand-writing a rule for each – plus, it only has to build a specific rule for combinations that it actually encounters. Combined with monotonic inferences, the prototypes can also support relational tests. For instance, to test that *location* was north of *destination* would require one production to elaborate working memory whenever the north relationship existed, and one numeric preference prototype to match against this elaboration.

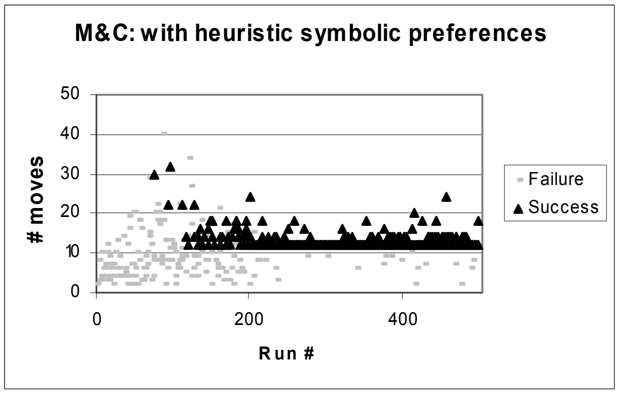
Results

What is detailed in the previous section is what has been implemented in Soar, and is available to any domain that is encoded in Soar. We tested the system in two domains. In both cases, we wrote Soar agents with little or no operator selection knowledge, and so relatively random initial behavior. We looked for improvement in their performance, relative to their reward functions, as they gained experience in the world. Only reinforcement learning was used – there was no chunking.

Missionaries & Cannibals: Our first test domain was the puzzle-problem Missionaries & Cannibals. To turn this into an RL problem, we gave the agent rewards of +1 for moving into a success state (all persons on right bank), -1 for moving into a failure state (majority cannibals on some bank), and 0 for other state transitions. In this world, working memory always contained a description of the current state in terms of the number of missionaries and cannibals on each bank and the location of the boat. Thus, the state was directly observable, and, since the state space was small, we could use the representation of one numeric preference rule for each state-operator pair. Below are the results of 500 training runs, in which the agent had no control knowledge other than its improving numeric preferences. In the graphs, black vs. gray indicates whether a run ended in success or failure, and the y-value measures the number of steps in the run.



By the end of these runs, the agent is generally successful. Mistakes are attributable to the persistent randomizing effect of a soft greedy policy. Learning can be accelerated considerably by the addition of heuristic symbolic preference rules. For instance, after modifying the Soar-RL agent above to give a symbolic *worst* preference to a move reversing the previous move, the results over 500 training runs were (note different y axis scale: 45 vs. 250):



Eaters: Puzzle problems are often used to demonstrate the speed-up effect of Soar's chunking mechanism, and they are useful for evaluation because the desired behavior is known. However, puzzles have a couple of faults as test domains for reward-based learning. First, they tend to offer rewards only as one-time marks of success or failure, rather than as a varied stream of feedback. Second, they frequently demand less interesting, more table-based representations of value functions, because their puzzling nature is often achieved by forcing the agent to consider all features in combination, rather than calculating the implications of each feature independently. Our second test domain attempts to avoid these faults. It is called Eaters and is played by a Pacman-like agent, called an eater, moving around a board.

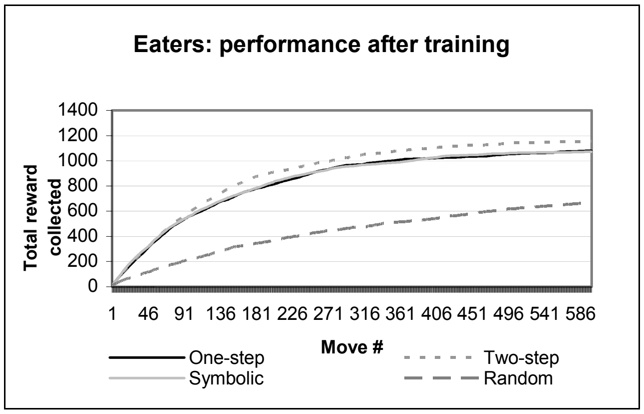
The Eaters board initially is filled with food of two types (bonusfood and normalfood) as well as some interior walls. An eater receives a reward of +10 for moving onto a bonusfood, +5 for moving onto a normalfood, and 0 for moving into an empty cell. (Contents are consumed when the agent moves onto them.) Additionally, there is a discount factor that favors earlier rewards. The agent can move in the four cardinal directions, if unblocked by a wall, and can sense only the contents of the cells in the 5x5 square centered on itself.

Given the agent's sensing capabilities, there are a variety of features of the state that could be represented – for instance:
1. The contents of one of the four adjacent cells.
2. The contents of all of the four adjacent cells.
3. The contents of the three new cells that could be visited on its second move given that its first move is in direction x.
4. The contents of the twenty-five cells that it can sense.

Of course, to build a numeric preference rule, a feature must be combined with an operator – for instance, the contents of the cell to the east *plus* move north or the contents of an adjacent cell *plus* an operator to move to that cell.

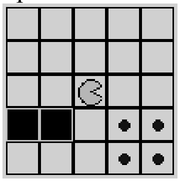
We tested two RL agents in this world. The first agent associated numeric preferences with the contents of the next cell; ultimately, it learned only three numeric preference rules – one each for **move to bonusfood**, **move to normalfood**, **move to empty**. The second agent had rules of this type, plus rules of type 3 in the list above; therefore its predicted value for a move was a sum of the immediate value associated with the move and a value associated with the resulting options for its second move.

These two agents were trained for 20 runs of 600 moves each and then their performance was tested. During testing, reinforcement learning was turned off and the parameter controlling randomness in operator choice was reduced, so that the agent had less of a tendency to explore. For comparison purposes, we also tested a random agent and a non-RL agent with symbolic preferences favoring **move to bonusfood** over **move to normalfood** over **move to empty**. The results are graphed below (total reward as a function of move #), where each line represents an average over twenty runs.



The major points demonstrated in this graph are

- Both trained RL agents have learned well enough to soundly beat the random agent.
- The performance of the one-step look-ahead agent is identical to that of the symbolic agent. The one-step look-ahead RL agent learned the values 50.12, 29.38, and 5.66 for **move to bonusfood**, **normalfood**, and **empty**, respectively, so it would be expected to have the same behavior as the symbolic agent.
- Adding the two-step look-ahead feature yields a relatively small but consistent gain in performance. The nature of this world is such that a move that looks best from the one-step perspective is generally the best move. The two-step look-ahead is most useful in deciding between moves when the one-step cell contents are the same. For instance, in the pictured situation, the eater has a numeric preference of 14.5 for all four **move to empty** operators, but a two-step preference of -6.6 for (**empty, empty, normalfood**) to the east versus -13.8 for (**empty,empty,wall**) to the west. The eater agent learned 56 two-step numeric preference rules vs. 3 one-step rules. These numbers are clearly out of proportion to the gains in performance; much of the needed control knowledge in this world can be captured by the one-step rules. Still, we avoided learning the 3x56 rules that would have been required if our two types of features had not been considered independently.



Comparison with ACT-R

ACT-R has a learning component that is comparable to reinforcement learning. The locus of selection in ACT-R is at the level of individual rule selection instead of at the level of operators in Soar. In ACT-R, only a single rule is fired at a time and the selection is done probabilistically based on a utility that is associated with each rule, with the selection distribution similar to the Boltzmann distribution. The utilities are a function of the probability of and expected cost of goal completion given that a particular production is chosen, as well as of the value of the current goal. The value of a goal is set by the programmer, and the other two numbers are learned from experience.

Below is a list of key differences and their possible implications between Soar-RL and ACT-R rule tuning:

1. Soar allows both symbolic and numeric preferences to contribute to the decision. Thus, it is possible to encode control knowledge that one option is better than another. There is no way to directly encode similar knowledge in ACT-R for preferring one rule to another.
2. The utilities captured by the numeric preferences in Soar do not rely solely on the idea of progress toward a goal, but instead are associated with a more flexible reward function. This reward function is capable of representing goal-directed activity, but can also be used for ongoing activities, such as the top-level of an agent, which monitors energy levels or selects goals to be pursued.
3. The utilities for a rule in ACT-R are related to a single goal, whereas in Soar, there can be many different rules for an operator that are sensitive to different goals. Thus, agents developed in ACT-R, may have difficulty learning in environments with shifting or multiple goals.
4. Ultimately, the utility of an action and its likelihood of being selected in ACT-R are based on the conditions of the individual rule. There is no run-time combination of the values as there is when the numeric indifferent preferences are combined in Soar. Thus, Soar distinguishes between the conditions for when an action/operator is valid and when it is desirable, which in turn gives it a richer representation for capturing complex reward/utility functions.
5. Although ACT-R can get the effect of combining multiple pieces of advice by using multiple rules with different conditions for the same actions, its learning mechanism can give credit to only one of these rules. In Soar-RL, all numeric preferences that contribute to the selection of an operator receive credit for the results of selecting that operator.

In this paper we have described a significant extension to the Soar architecture – the addition of reinforcement learning. This provides a new, architectural approach to learning control knowledge based on expected rewards. The implications for Soar as an AI architecture are clear – all Soar programs will now be able to automatically adjust to feedback from their environments in cases where symbolic preferences are inadequate to make decisions. The impact on Soar for cognitive modeling is less clear, but intriguing. Soar-RL, together with recent work in adding activation to Soar (Nuxoll, Laird, & James, 2004), moves Soar much closer to ACT-R in terms of combining numeric and symbolic computation and learning. The underlying mechanisms share many common features, but also differ enough that they might lead to different computational models of human behavior. Even if these models do not surpass current ACT-R theory, they should provide us with new insights into what aspects of the ACT-R models are most important.

Future work

For future work, we have two basic thrusts. One is to build models of human performance, comparing and contrasting these models with existing ACT-R models. The second thrust, is to greatly expand the use of reinforcement learning in Soar to be more complete, which will push our implementation in directions that have not yet been explored in other cognitive architectures.

1. Task independent learning parameters. Reinforcement learning invariably has parameters that are used to tweak learning for a specific problem. In addition, we have a Boltzmann decision procedure that has a temperature parameter that influences the evenness of probabilistic selection. Although there may be truly task-independent values for these parameters, embedding the decision procedure and learning mechanism in a general architecture allows us to explore the possibility that these parameters can be tied to a general feature of the current task that is detectable by the architecture. For example, temperature works best in the decision procedure if its value is high during earlier explorations of a domain so that exploration is preferred to exploitation. As the domain is explored and the knowledge matures, exploitation is usually preferred, so that temperature can be lowered. As demonstrated in adaptive simulated annealing (Ingber 1993), dynamically adjusting the temperature can greatly improve performance. To begin with, we will derive the temperature from the certainty of the numeric preferences. When the variance is high, the temperature will also be high leading to a more uniform distribution of the decision space and encouraging exploration, but when the variance is low, the temperature will be low, biasing the decision to those operators with higher numeric preference.
2. Learning the appropriate features to associate with predicted rewards. The state spaces that Soar works in are structured (involving relations) and huge, so that many standard approaches to representing the association between features and state values are inadequate. We use rules for such associations, but that then begs the question as to where those rules come from. We plan to investigate schemes where rules are dynamically generated by specializing existing rules.
3. Hierarchical Learning. Soar automatically generates subgoals. This gives us the opportunity for the system to learn about not only the overall task it is trying to achieve, but also the component parts of the task: subtasks and meta-reasoning. These components are more likely to be shared across many different tasks, so that learning about subgoals can improve performance on novel tasks with common subtasks. Thus, an important issue is to determine how reinforcement learning should work across subgoals, with the main issue being how to introduce rewards at the end of a subtask. These rewards should be related to subgoal achievement, and may be independent of environmental rewards. This is the approach taken in several hierarchical RL systems, in particular MAXQ (Dietterich, 2000) and options (Sutton, Precup, & Singh 1999), but neither system explains how these subgoal rewards may be generated automatically. For an autonomous agent, the origin of those reward functions is unclear because subgoals may not be preprogrammed but arise from a lack of knowledge.
4. Internal, task-independent reward functions. Our goal is to integrate reinforcement learning into Soar so that it can be used on any task. One key component of reinforcement learning is obtaining a reward from the environment. However, for many tasks, the environmental rewards are extremely sparse, making learning very slow. We want to investigate domain-independent reward functions that augment external reward functions and help steer an autonomous agent to useful parts of its environment – these are almost meta-rewards in that they are based on characteristics of the knowledge that has been learned, emphasizing achieving improvements in the agent's ability to predict its environment (Kaplan 2003). Emotions may be an alternative (or possibly related) source of reward that can direct behavior.

References

Anderson, J. R. & Lebiere, C. (1998). *The Atomic Components of Thought.* Mahwah, NJ: Erlbaum.

Dietterich, T. (2000). Hierarchical Reinforcement Learning with the MAXQ Value Function Decomposition. *Journal of Artificial Intelligence Research*, 13.

Ingber, L. (1993) Simulated Annealing: Practice versus Theory, Mathematical Computer Modeling, 16 (11), 29-57.

Kaplan, F. (2003) Bootstrapping Awareness, In Dautenhan, K. and Nehaniv, C., editor, Proceedings of Second International Symposium on Imitation in Animals and Artifacts. 48-57.

Newell, A. (1990). *Unified Theories of Cognition*. Cambridge, MA: Harvard University Press.

Nuxoll, A., Laird, J., James, M. (2004). Comprehensive Working Memory Activation in Soar..International Conference on Cognitive Modeling.

Rummery, G. A., & Niranjan, M. (1994). On-line Q-Learning using Connectionist Systems. Technical Report CUED/F-INFENG/TR 166. Engineering Department, Cambridge University.

Sutton, R. & Barto, A. (1998). *Reinforcement Learning: An Introduction.* Cambridge, MA: The MIT Press.

Sutton, R., Precup, D., & Singh, S. (1999). Between MDPs and Semi-MDPs: A Framework for Temporal Abstraction in Reinforcement Learning. *Artificial Intelligence*, 112(1-2).

Wray, R., & Laird, J. (2003). Variability in Human Behavior Modeling for Military Simulations. Proceedings of the 2003 Conference on Behavior Representation in Modeling and Simulation. Scottsdale, AZ.