Deconstructing Reinforcement Learning in Sigma

Paul S. Rosenbloom

Department of Computer Science & Institute for Creative Technologies
University of Southern California
12015 Waterfront Drive, Playa Vista, CA 90094
rosenbloom@usc.edu

Abstract. This article describes the development of reinforcement learning within the Sigma graphical cognitive architecture. Reinforcement learning has been deconstructed in terms of the interactions among more basic mechanisms and knowledge in Sigma, making it a derived capability rather than a de novo mechanism. Basic reinforcement learning – both model-based and model-free – are demonstrated, along with the intertwining of model learning.

Keywords: Reinforcement learning, cognitive architecture, graphical models.

1 Introduction

Reinforcement learning (*RL*) enables agents to learn effective policies for task performance based on rewards received over a sequence of trials [1]. It is a key concept in artificial general intelligence (AGI) – even being at the core of a proposal for a *universal artificial intelligence* [2] – plays an important role in intelligent robotics, and is increasingly important in conventional cognitive architectures [3-4]. This article describes the simple manner in which RL can be implemented within the *Sigma* (Σ) cognitive architecture [5], with its grounding in *factor graphs* [6] – a general form of graphical model [7] – and *piecewise linear functions* [8].

The goal of this effort has not been to implement from scratch a preselected RL algorithm within Sigma, nor even necessarily, at least at first, to yield an RL capability that is competitive with today's best, but to: (1) explore whether some variant of RL could emerge from how Sigma already works, and (2) analyze the ensuing results to see what they can tell us about both Sigma and RL. This approach to RL is driven by a key desideratum that is guiding Sigma's development towards general intelligence – *functional elegance*, which seeks to combine the broad range of capabilities implicit in general intelligence with simplicity and theoretical elegance. The ultimate aim is for something like a set of *cognitive Newton's laws* that yield the required diversity of behavior from interactions among a small set of very general primitives. AIXI [2] can be viewed as an attempt at an extreme example of functional elegance. The approach in Sigma is less ambitious, but still strongly in this direction.

This article explains how model-based RL can be engendered within Sigma from the interactions among: (1) a more primitive gradient-descent learning mechanism that is capable, among other things, of learning to predict; and (2) schematic knowledge that determines what predictions are to be learned, what their initial values should be, and how to propagate such values backwards over time. This effectively deconstructs a form of model-based RL in terms of preexisting, more basic, capabilities already in Sigma, plus knowledge. In contrast, no means was found within Sigma's existing capabilities of producing either model-free RL or the intertwining of model learning with model-based RL. However, both do become possible after a minimal further addition to the architecture. This overall approach, of deconstructing capabilities in terms of existing architectural mechanisms when possible, and of minimal changes to the architecture only when necessary, directly supports functional elegance. It also reflects both a form of Occam's razor and an adherence to Allen Newell's exhortation to "listen to the architecture" [9].

2 Reinforcement Learning (RL)

The central concept in reinforcement learning is that of (logically) propagating rewards received later in performance backwards in time to assist in learning the expected utility of earlier actions (for use in later trials). Ultimately the learning is reflected in *Q values* – $Q(s, a)$ – which capture the expected (discounted) cumulative reward of choosing action a in state s, and which thus aid in selecting appropriate actions. The particular approach taken in Sigma provides an *on-policy* learning algorithm, which learns from the action taken rather than from the best action that could have been taken, making it more akin to SARSA [10] than to Q-learning [11]. The learning update in SARSA is defined as $Q(s_t, a_t) \leftarrow Q(s_t, a_t) + \alpha[r_t + \gamma Q(s_{t+1}, a_{t+1}) - Q(s_t, a_t)]$, where α is the learning rate and γ is the discount factor for future rewards.

Consider, for example, a one-dimensional, discrete, grid task in which the agent may start at any location and is to reach a goal location via `left` and `right` actions (Fig. 1). With no initial information concerning which operator to choose, behavior begins with random choices. However, once the goal location is reached, a reward will be received, and learning can begin. Over time, and future experiences, this information propagates backwards across actions to yield Q values that predict higher discounted cumulative rewards for choosing `right` when the agent is to the left of the goal and `left` when it is to the right of the goal. This example task will be used throughout the remainder of this article.

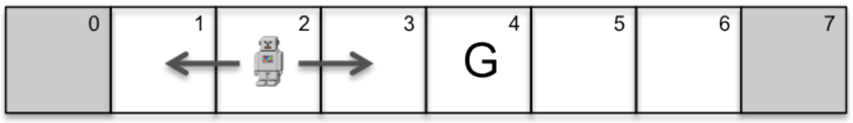

Fig. 1: 1D grid task with example goal (4), starting location (2), and actions (`left` and `right`). The two extreme locations act as buffers to avoid end effects.

3 The Sigma Architecture

Sigma has been under development in some form since 2008, although until now it lacked a name due to an ambivalence concerning whether what was being developed was a specific graphical architecture or a general approach, based on graphical models, for exploring the space of architectures. Although there remains room to explore a broader range of architectures, it has become increasingly clear that a specific architecture was being built, which now has a proper name: *Sigma*.

In general, graphical models provide an efficient means of computing with complex multivariate functions by decomposing them into products of simpler functions and then mapping them onto graphs. From these graphs, the marginals of the individual variables – i.e., the function's values when all other variables are summarized out – can be computed efficiently, as can the function's global mode. Bayesian networks and Markov random fields are common forms of graphical models, and some forms of neural networks map directly onto them. Factor graphs are a variant of graphical models that map decompositions of arbitrary multivariate functions onto undirected bipartite graphs of variable and factor nodes. Variables map onto variable nodes while decomposed factors map onto factor nodes. Undirected edges are defined between each factor node and its variables. Fig. 2 shows a factor graph for a simple multivariate algebraic function, along with its solution via the summary product algorithm [6], as is used in Sigma.

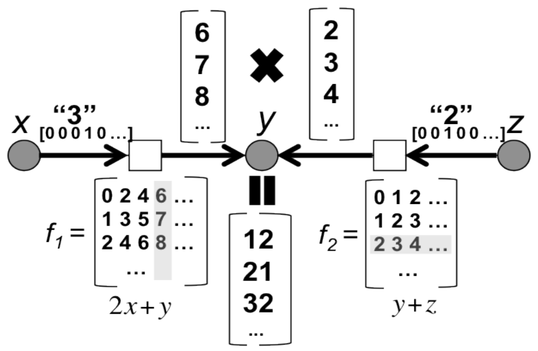

Fig. 2: Summary product computation over the factor graph for $f(x,y,z) = y^2+yz+2yx+2xz = (2x+y)(y+z) = f_1(x,y)f_2(y,z)$ of the marginal on y given evidence concerning x and z. Only the messages (and link directions) involved in computing y are shown.

Given evidence about a subset of the variables, messages are passed along the links and processed at the nodes to yield new messages. Each message along a link provides information about the distribution of values for the link's variable. Incoming messages at variable nodes are combined via pointwise product – like an inner product without the final summation – to yield outgoing messages, but with each outgoing message omitting from its product the incoming message on its link. Similar pointwise products occur at factor nodes, but with the factor's function also included in the product; and then all variables not in the outgoing message are summarized out. Summarization typically occurs via *summation* – or *integration* for continuous functions – to yield marginals, or via *maximum* to yield the mode. Message passing terminates when a stopping criterion is hit, such as that no new message is significantly different from the previous message along the same link.

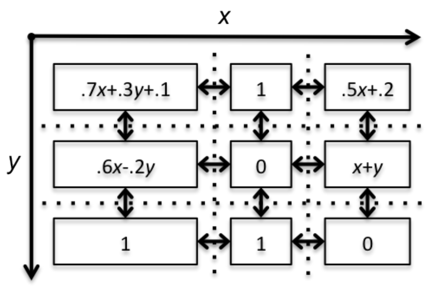

Fig. 3: Bivariate function as a 2D array of regions with linear functions.

The generality and efficiency of the summary product algorithm depends critically on the representation used for the factor functions and messages. In Sigma, a multidimensional piecewise linear representation is used, with one dimension per variable (Fig. 3) [8]. This enables approximating arbitrary continuous functions as closely as desired, plus specialization to discrete representations – such as probability distributions – by mapping integers in the function's domain to unit regions while limiting the region functions to constants, and to symbolic representations by further limiting the constant functions to Boolean (0/T and 1/F) while assigning symbols to domain integers. A form of *hybrid mixed* representation is thus proffered.

Knowledge fragments in Sigma are specified via *conditionals*, such as the one in Fig. 4, which compile into subgraphs of long-term memory. What is normally viewed as evidence in graphical models appears in working memory nodes in Sigma. The conditional in Fig. 4 consists of two conditions and an action, thus amounting to a classical rule.

```
CONDITIONAL Move-Left
   Conditions: (Selected state:s operator:left)
               (Location state:s x:x)
   Actions: (Location state:s x:x-1)
```

Fig. 4: Grid conditional for executing action of moving left.

The expression `x-1` in the conditional's action indicates the use of an *offset* [12], part of Sigma's mechanism for *affine transformations* (in support of mental imagery) [13]. In general, a variable in a condition or an action may include a coefficient and an offset, where the coefficient must be a constant and the offset may be either a constant or a variable. This isn't simply a matter of multiplication and addition of values though, as an offset shifts a whole piecewise linear function along a variable's dimension by modifying the region boundaries, while a coefficient may – once again by modifying region boundaries – expand, contract, or invert a dimension. The combination of coefficients and offsets enables mental imagery to be translated, scaled and reflected. When combined with variable interchanges, they also enable limited forms of rotation.

When the offset is a variable rather than a constant, two random variables must be added, implicating a *convolution* in general. Although convolutions have not yet been implemented in Sigma, when the offset variable only has a single nonzero value, it can simply be extracted and used like a constant. Such an approach is exploited in RL to add the current reward to the (distribution over the) discounted future reward.

Another feature of Sigma that is relevant to the implementation of RL is a generalization from the use of constants in conditions and actions – such as `left` in Fig. 4 – to the use of *filters*. A constant in this context is essentially a filter that only passes through portions of messages that match it via a factor function that is nonzero only for the constant. This has been generalized to allow arbitrary piecewise linear

```
CONDITIONAL Select-Operator
   Conditions: (Location state:s x:x)
               (Q x:x operator:o value:[.1*q])
   Actions: (Selected state:s operator:o)
```

Fig. 5: Grid conditional that transforms distributions over Q values into operator weights for selection.

functions to appear where previously only constant tests could. Fig.5, for example, shows a conditional with a filter – in square brackets to distinguish it from an affine transform – that converts distributions over the possible Q values for the operators, ranging in [0, 10), into an expected Q value for each operator. Q's domain values are multiplied by .1, with the result then multiplied by the incoming message. The variable *q* is summarized out via integration prior to the action, weighting each operator by its expected Q value.

Conditions and actions in Sigma limit the direction in which messages are passed – those within condition subgraphs only move away from working memory while those within action subgraphs only move towards it. This provides the forward momentum central to procedural memory. *Condacts* – a neologism for *cond*itions and *act*ion*s* – provide the bidirectional message passing required for the full generality of factor graphs, as used for example in probabilistic reasoning, constraint satisfaction, signal processing, and (partial match in) declarative memory [14]. The conditional in Fig. 6 defines a transition function – i.e., an action model – using two conditions, a condact, and a function to specify an initial uniform distribution over the next location given the current location and operator. The stars (*) in the function denote that the value specified (.125) applies to all triples of current location, selected operator, and next location. The variable *nx* for the next state is underlined to denote normalization over it during learning.

```
CONDITIONAL Transition
   Conditions: (Location state:s x:x)
               (Selected state:s operator:o)
   Condacts: (Location*Next state:s x:nx)
   Function<x,o,nx>: .125:<*,*,*>
```

Fig. 6: Grid conditional for an initially uniform transition function (action model).

The core cognitive (or *decision*) cycle in Sigma involves message passing until quiescence, with the results then used in deciding how to modify working memory. Learning also occurs at decision time, by altering functions in conditionals (structure learning remains for future work). *Episodic learning* modifies temporal functions in episodic conditionals that are automatically built for state predicates (such as `Location` and `Selected`). *Gradient descent learning* modifies conditional functions, as stored in factor nodes, by interpreting incoming messages as gradients that are to be normalized, multiplied by the learning rate, and added to the existing function. The idea for this learning mechanism, which was developed in conjunction with Abram Demski and Teawon Han, was inspired by earlier work [15] showing that gradient descent was possible in Bayesian networks, much as in neural networks, but without the need for an additional backpropagation mechanism because the local messages already determined the gradient.[1] This form of learning is capable of

[1] The version here only approximates the true gradient in [15], but was sufficient for this work.

working in either a supervised or unsupervised manner, and in Sigma supports both basic RL and model learning.

4 RL in Sigma

The core idea for deriving an RL algorithm from Sigma has been to leverage gradient descent in learning Q values over multiple trials, given appropriate conditionals to structure the computation as is needed for this to happen. Much of the work has therefore involved understanding what these conditionals should be.

Two conditionals – the one in Fig. 4 plus another like it – implement the `left` and `right` actions in the grid task. Given these two conditionals, plus a third that proposes the actions for selection, Sigma performs a random walk until the goal is achieved. To enable Q values to determine which action to choose, the proposal conditional must be augmented to use them as operator weights – or *numeric preferences* – as in Fig. 5. Initial Q values must then also be provided, as in Fig. 7. If direct evidence were provided for the action's Q values, it would be trivial to use gradient descent to learn better values for this function without needing to invoke reinforcement learning. However, without such evidence, RL is the means by which rewards from later steps in task performance propagate backwards to serve as input for learning Q values for earlier steps. This occurs via a combination of: (1) learning to predict local rewards from the externally provided evidence for these rewards; and (2) learning to predict both discounted future rewards and Q values by propagating backwards the discounted sum of the next location's local reward and its discounted future reward.

```
CONDITIONAL Q
   Conditions: (Location state:s x:x)
   Condacts: (Q x:x operator:o value:q)
   Function<x,o,q>: .1:<*,*,*> …
```

Fig. 7: Grid conditional for an initially uniform distribution over the Q values for the operators, given the locations.

```
CONDITIONAL Reward
   Condacts: (Reward x:x value:r)
   Function<x,r>: .1:<[1,6)>,*> …
```

Fig. 8: Grid conditional for an initially uniform distribution over rewards at locations.

To (learn to) predict a location's reward, the conditional in Fig. 8 is added. To learn discounted future rewards and Q values, the conditional in Fig. 9 is added (along with

```
CONDITIONAL Backup
   Conditions: (Location state:s x:x)
               (Selected state:s operator:o)
               (Location*Next state:s x:nx)
               (Reward x:nx value:r)
               (Projected x:nx value:p)
   Actions: (Q x:x operator:o value:.95*(p+r))
            (Projected x:x value:.95*(p+r))
```

Fig. 9: Grid conditional for backing up rewards.

an unshown conditional for discounted future rewards). The Backup conditional examines the current location and operator, along with the predicted next location – as given by the transition function – and its predicted local reward and future discounted reward. In the actions, it leverages an affine transformation, with an offset to add the next location's predicted local reward to the distribution over its predicted future reward, and a coefficient to discount this sum. RL then results from using the messages that are passed back to the conditional functions as gradients in learning Q values and discounted future rewards.

Fig. 10 summarizes how RL emerges from all of this. Double arrows with elliptical tips represent decisions for the operator and location. Solid arrows predict aspects of the current location. The gray box is the external reward. Dotted boxes and arrows are predictions of/for the next location. Value backup involves the gray triangles and curved arrow.

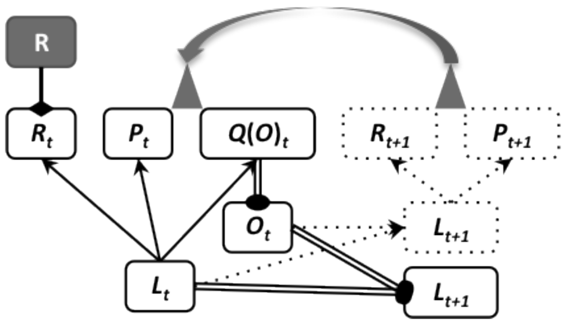

Fig. 10: Variables and processes for RL in the grid task.

The resulting form of learning is like SARSA rather than Q-learning because it is driven by the operator actually selected rather than by the best available operator. This form of RL also is *model based*, leveraging a version of the transition conditional in Fig. 6 that embodies probabilities corresponding to the actions' actual effects. Learning then occurs via gradient-descent-based refinements to the functions in Figs. 7-8 and the unshown one, for the distributions over Q values, local rewards, and discounted future rewards, respectively.

After completing 20 trials for each of the two possible extreme starting points – locations 1 and 6 – the expected value of the learned reward function (by location) is identical to the externally defined reward function: <0, 0, 0, 0, 9, 0, 0, 0, 0>. The expected values learned for the discounted future reward are shown in Fig. 11 (Fixed Model). This peaks, as it should, as the goal location (4) is neared, but is zero for both the goal location and the buffer locations since

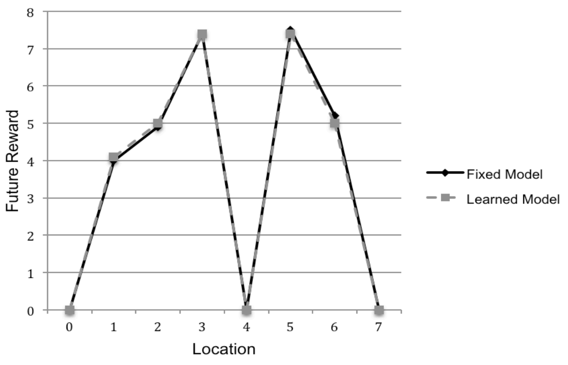

Fig. 11: Learned expected discounted future reward.

they are initialized to zero and no move is ever made from them. The expected Q values learned for `left` vs. `right` are shown in Fig. 12. As desired, moving right is preferred when left of the goal and moving left when to the right. There is no preference at the goal.

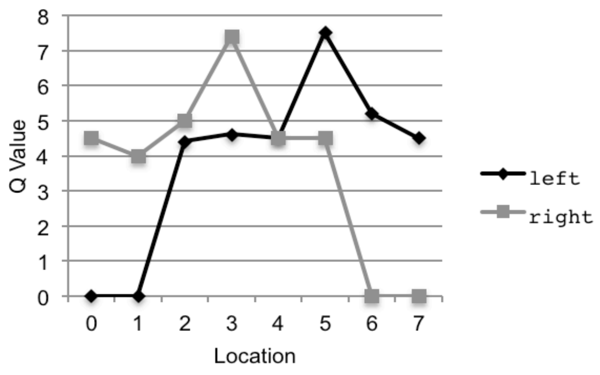

Fig. 12: Learned expected Q values.

These results have been presented in terms of point values, a format that matches what is normally seen with RL. However, the learning actually involves full distributions rather than individual points, with points computed as expected values over distributions. Learning via distributions rather than points has been natural in Sigma, but it may also prove particularly advantageous when distributions can help, for example, identify when a representation is too coarse [16], or when a Soar-like impasse – forms of which already exist in Sigma [17] – should occur [18].

Everything in this example was learned in a *synchronic* manner, considering only one actual location. Even reward backup was synchronic, being based on the distribution over the predicted next location rather than on the actual next location. By focusing on learning to predict, RL has been able to proceed within Sigma in the context of a single actual location. However, for model-free RL, a pair of actual locations must be available simultaneously in working memory so that value backup can occur without the aid of the predictions the transition function provides in model-based RL. Similarly, although an initial uniform transition function is provided when the action models are to be learned, the correct gradient cannot be computed unless both locations are simultaneously in working memory.

As Sigma worked prior to this investigation of RL, consecutive states were simultaneously present only during the decisions that occurred at the end of cognitive cycles, when old working memory values were replaced by new ones. However, just one of these states would be in working memory at a time. If Sigma were extended to transiently represent both at once in working memory – essentially during the decision – with a solution to the graph occurring in the interim and learning enabled, then the kind of *diachronic* learning required for both model-free RL and the learning of action models should be possible with only a minimal extension to Sigma's architectural code. This is in fact what has been implemented. During decisions, new values are placed into *next* variants of to-be-altered state predicates – `Location*Next` here – and the graph is again solved with learning enabled, before actual modifications are made to working memory (and the *next* variants are flushed).

Now, when there is no transition conditional, model-free RL results, with value backup based on the actual next location rather than the predicted one. Given 20 trials, the expected discounted future rewards are the same as those learned with a fixed model (Fig. 11). When the uniform transition conditional from Fig. 6 is included, the gradient necessary to learn action models becomes available, enabling them to be acquired during the same trials in which rewards, Q values, and discounted future rewards are learned. Running 20 trials here yields a transition function where the only entries that are above the initial value of .125 are shown in Fig. 13 (with darkness corresponding to functional value). All of the on-path moves have a functional value of 1, whereas the two off-path moves predict the correct transition but at lower values. The expected discounted future rewards here – Fig. 11 (Learned Model) – are nearly indistinguishable from those learned with a predefined transition function.

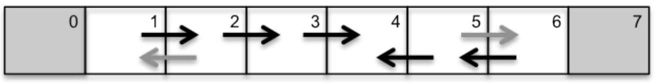

Fig. 13: Learned transition function.

5 Conclusion

Learning is central to general intelligence, with reinforcement learning providing a particular form that that has been prominently featured within both AGI and several cognitive architectures. When the time came to address how reinforcement learning would work in Sigma, the intriguing possibility arose of its emerging from the interactions among a general set of more basic mechanisms, making RL a derived capability rather than an architecturally implemented mechanism, and satisfying the joint constraints of functional elegance, Occam's razor and Newell's exhortation.

The work presented here is still only a beginning, but it does show how RL can be deconstructed in terms of a local form of gradient-descent learning plus appropriate knowledge structures, to yield basic on-policy, model-based, reinforcement learning. A single extension to Sigma – to simultaneously represent both the current and next state during an interpolated graph solution – was then required to enable both model-free RL and (intertwined) model learning. As it turns out, this is a non-RL-specific extension that was also motivated, for example, by the related problem of learning transition functions for POMDPs in Sigma [19]. The extension of Sigma's affine transformations to variable offsets also occurred in service of implementing RL, although the idea and the understanding of its need both predated this work on RL.

Much more is still required in a complete, state-of-the-art, architecturally integrated capability for reinforcement learning, including exploration, scaling, and structure learning. Also necessary is extensive experimentation with more complex tasks, careful comparisons with implementations of RL in other architectures, and investigations of synergies that might become available when RL interacts with other knowledge and capabilities in Sigma. Yet, the important result remains, that the core of RL has been demonstrated, along with its intertwining with model learning, and all in a functionally elegant manner.

Acknowledgments. This effort has been sponsored by the U.S. Army and the Air Force Office of Scientific Research. Statements and opinions expressed do not necessarily reflect the position or the policy of the United States Government, and no official endorsement should be inferred. I would like to thank Abram Demski, Nassim Mafi and Volkan Ustun for helpful discussions on this material.

References

1. Sutton, R.S, Barto, A.G.: Reinforcement Learning: An Introduction. A Bradford Book, MIT Press, Cambridge (1998)
2. Hutter, M.: Universal Artificial Intelligence: Sequential Decisions Based on Algorithmic Probability. Springer-Verlag, Berlin (2005)
3. Sun. R., Slusarz, P., Terry, C.: The interaction of the explicit and the implicit in skill learning: A dual-process approach. Psychological Review, 112, 159-192 (2005)
4. Nason, S., Laird, J.E., Soar-RL: Integrating reinforcement learning with Soar. Cognitive Systems Research, 6, 51-59 (2005)
5. Rosenbloom, P.S.: Graphical models for integrated intelligent robot architectures. In: AAAI Spring Symposium on Designing Intelligent Robots (2012)
6. Kschischang, F. R., Frey, B. J., Loeliger, H.: Factor Graphs and the Sum-Product Algorithm. IEEE Transactions on Information Theory, 47, 498-519 (2001)
7. Koller, D., Friedman, N.: Probabilistic Graphical Models: Principles and Techniques. MIT Press, Cambridge (2009)
8. Rosenbloom, P.S.: Bridging dichotomies in cognitive architectures for virtual humans. In: AAAI Fall Symposium on Advances in Cognitive Systems (2011)
9. Newell, A.: Unified Theories of Cognition. Harvard University Press, Cambridge (1990)
10. Rummery, G.A., Niranjan, M.: On-line Q-learning using connectionist systems (1994)
11. Watkins, C.J.C.H.: Learning from Delayed Rewards. PhD thesis, Cambridge University (1989)
12. Rosenbloom, P.S.: Mental imagery in a graphical cognitive architecture. In: Second International Conference on Biologically Inspired Cognitive Architectures (2011)
13. Rosenbloom, P.S.: Extending mental imagery in Sigma. In: Fifth Conference on Artificial General Intelligence (2012)
14. Rosenbloom, P.S.: Combining Procedural and Declarative Knowledge in a Graphical Architecture. In: 10th International Conference on Cognitive Modeling (2010)
15. Russell, S., Binder, J., Koller, D. Kanazawa, K.: Local learning in probabilistic networks with hidden variables. In: 14th International Joint Conference on AI (1995)
16. Munos, R., More, A.: Variable resolution discretization in optimal control. Machine Learning, 49, 291-323 (2002)
17. Rosenbloom, P.S.: From memory to problem solving: Mechanism reuse in a graphical cognitive architecture. In: Fourth Conference on Artificial General Intelligence (2011)
18. Bloch, M.K., Laird, J.E. Heuristic value function revision. In: The 32nd Soar Workshop.
19. Chen, J., Demski, A., Han, T., Morency, L-P., Pynadath, P., Rafidi, N., Rosenbloom, P.S.: Fusing symbolic and decision-theoretic problem solving + perception in a graphical cognitive architecture. Second International Conference on Biologically Inspired Cognitive Architectures (2011)