From Memory to Problem Solving: Mechanism Reuse in a Graphical Cognitive Architecture

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 extension of a memory architecture that is implemented via graphical models to include core aspects of problem solving. By extensive reuse of the general graphical mechanisms originally developed to support memory, this demonstrates how a theoretically elegant implementation level can enable increasingly broad architectures without compromising overall simplicity and uniformity. In the process, it bolsters the potential of such an approach for developing the more complete architectures that will ultimately be necessary to support autonomous general intelligence.

Keywords: Cognitive architecture, graphical models, memory, problem solving.

1 Introduction

A *cognitive architecture* is a hypothesis about: (1) the fixed mechanisms underlying intelligent behavior, and (2) how they integrate together in support of autonomous general intelligence. The ideal cognitive architecture would combine *broad applicability* – whether in terms of the span of natural phenomena covered or the range of artificial functionality produced – with *theoretical elegance* (uniformity and simplicity). But there is an inherent tension between these two characteristics; the former favors mechanism proliferation while the latter discourages it. The resulting *diversity dilemma* is one of the central issues in architectures [1]. How researchers respond to it determines much about the nature of the architectures they produce; consider, for example, the contrast between the eclectic approach in OpenCogPrime [2] and the more theoretically elegant approach in AIXI [3].

One recent approach seeks to build a diversity of architectural capabilities – for memory, decisions, learning, etc. – from the interactions among a small set of general mechanisms at the *implementation level* beneath the architecture [1]. Broad applicability at the architecture level is thus joined with theoretical elegance at the implementation level. *Graphical models* [4] were proposed as a basis for the implementation level because they yield state-of-the-art algorithms across symbol, probability and signal processing from a uniform representation and reasoning

To appear in *Proceedings of the Fourth Conference on Artificial General Intelligence* (AGI), 2011. The original publication is available at www.springerlink.com.

algorithm. They raise the possibility of uniformly implemented and tightly integrated architectures capable of spanning from perception to cognition and back to action.

An initial fragment of this potential was realized with the implementation of a *graphical memory architecture* that combined rule-based procedural knowledge, semantic and episodic declarative knowledge, and constraint knowledge (which blends aspects of both procedural and declarative knowledge) [5]. The first three were modeled on memories in the Soar architecture [6] and ideas from the ACT-R community [7]. The fourth was added simply because it came along essentially for free. This graphical memory architecture exploited the uniform combination of symbolic and probabilistic reasoning enabled by graphical models, and that is now at the core of the burgeoning subfield of *statistical relational AI*. It also supported continuous quantities, although performing no actual signal processing.

Ongoing work is extending this partial architecture to include problem solving, reflection, learning, and mental imagery; all in service of a medium-term goal of a uniformly implemented *hybrid* (discrete + continuous) *mixed* (symbolic + probabilistic) variant of Soar, and a long-term goal of theoretically elegant yet broadly applicable architectures. This article presents results from extending the memory architecture to incorporate basic internal problem-solving capabilities, based on Soar, with a particular emphasis on how such problem solving is supported by general mechanisms already implemented in service of the memory architecture. The resulting contributions are fourfold: (1) the extension of the graphical memory architecture to problem solving; (2) an evaluation of the generality of the graphical implementation mechanisms with respect to how well they extend from memory to problem solving; (3) presentation of heretofore unpublished aspects of the graphical memory architecture and its implementation that are important for understanding the first two contributions; and (4) an approximate reimplementation of key aspects of the Soar architecture with enhanced uniformity and elegance at the implementation level.

2 Problem Solving in Soar

The heart of problem solving is the *selection* and *application* of operators that perform internal actions and control or simulate external actions. Selection requires generation and comparison of candidate operators and then a choice among them. For both internal actions and simulations of external actions, application requires changing the internal state to correspond to the operator's effects. Control of external actions requires both perception and motor control. As perception and motor control are beyond the scope of this article, the focus here is restricted to internal problem solving via internal actions and simulations of external actions.

Soar represents the state of problem solving in a symbolic working memory (WM). Generation of candidate operators occurs via retrieval from long-term memory (LTM) into WM, as cued by the contents of the state (including any current goals). Then, based on the state and the proposed operators, preference information respecting operator selection – whether symbolic or numeric – is also retrieved from LTM. Except for acceptable preferences, which propose operators for selection, retrieved preferences are maintained outside of WM, in a separate preference memory (PM).

Preference memory is normally omitted from descriptions of Soar, as it is considered an implementation detail rather than part of the theory, yet it is an important and distinct form of memory that was added specifically in support of problem solving. Operator selection is based on the contents of PM plus a separately encoded decision procedure. Once an operator is selected, state changes are retrieved from LTM – based on the operator and the state – engendering modifications to working memory. This combination of capabilities for operator selection and application comprises what can be called *base-level problem solving* in Soar. Soar can also engage in *meta-level problem solving*, where the inability to select a new operator yields an impasse plus a meta-level state in which the impasse can be resolved via reflection [8]. However, reflection is a large enough topic in its own right that discussion of its graphical implementation has been deferred to a follow on article.

Base-level problem solving in Soar is normally viewed as occurring via two nested loops: (1) the *elaboration cycle*, in which all legal instantiations of all rules fire (logically) in parallel, yielding one round of changes to working memory; and (2) the *decision cycle*, comprising an elaboration phase during which elaboration cycles repeat until quiescence – i.e., until no further rules can fire – followed by a call to the decision procedure and the resulting selection of an operator in working memory. However, there is actually one additional cycle that is nested within the elaboration cycle: (0) the *match cycle*, in which tokens representing intermediate match results are passed around within the Rete network [9]. As with preference memory, this is considered an implementation detail in Soar rather than as part of the theory.

Retrievals from long-term memory for operator selection remain active – in WM or PM – only while their triggering conditions are valid. Thus, as the state changes, candidate operators and preferences automatically retract – in a manner akin to truth-maintenance systems – when they become inapplicable. In contrast, retrievals from long-term memory for operator application remain active until explicitly removed. This provides an implicit frame axiom, retaining all aspects of the state not explicitly changed. The distinction between selection and application knowledge effectively yields a problem-solving-driven partitioning of Soar's single rule memory into two procedural memories that differ both in when they are used during problem solving and in how their results are maintained over time.

Memory plays a critical role in Soar's problem solving, through storing, retrieving and maintaining states, operators and preferences. This amounts to a significant bit of architectural capability reuse, from WM and LTM to problem solving, and is the kind of gain Soar has long featured from integration across its capabilities. But there is no finer-grained sharing of mechanisms at the implementation level. For example, the Rete match mechanism at the heart of Soar's procedural memory is not reused in its declarative memories. Nor is it leveraged to implement the PM or decision procedure necessary for problem solving. It simply isn't a general enough implementation mechanism to do more than the one job it currently does extremely well.

If Soar's procedural memory were partitioned into two rule-based memories, by when and how the knowledge is used in problem solving, Rete could theoretically be reused across these two memories. But that would still be about it. Disjoint code implements memory (WM and LTM) versus problem solving (PM and the decision procedure); and, even within LTM, disjoint code implements rule, semantic and episodic memories. The latter disjointness was addressed earlier via general graphical

implementation mechanisms that supported a unified long-term memory containing both procedural and declarative knowledge. Here, we further build upon these same mechanisms to address the disjointness between memory and problem solving.

3 The Graphical Memory Architecture

The graphical memory architecture is based on running the *summary product algorithm* over *factor graphs* [10]. Factor graphs are similar to Bayesian and Markov networks, except that: unlike Bayesian networks, but like Markov networks, they employ bidirectional links between nodes; and, unlike both forms of networks, factor graphs explicitly include not only variable nodes but also factor nodes for functions over sets of variables. Factor graphs enable efficient computation with complex multivariate functions – whether representing probability distributions or arbitrary functions – by decomposing them into products of simpler functions and then mapping these decompositions onto graphs. By passing messages between variable and factor nodes concerning the possible values of variables, the summary product algorithm computes marginals on the variables (when using *sum* as the summarization operator), as well as computing more global properties such as maximum a posteriori (MAP) probabilities (when using *max* as the summarization operator).

Knowledge in long-term memory consists of generalized *conditionals* that can embody *conditions*, *actions*, *condacts* and *functions*. Figs. 1 and 2 show two examples. The first combines conditions and actions in a rule that avoids Eight Puzzle operators that move tiles from their goal locations. The second is a fragment of semantic memory that combines conditions, condacts and a function to represent the conditional probability of an object's weight given its concept.

```
CONDITIONAL GoalReject
   Conditions: (Operator id:o state:s x:x y:y)
               (Goal state:s x:x y:y tile:t)
               (Board state:s x:x y:y tile:t)
   Actions: (Selected - state:s operator:o)
```

Fig. 1: Eight Puzzle heuristic that rejects from consideration operators that move tiles out of place.

Conditions and actions are just like in traditional rules; conditions match to working memory elements and actions modify them. Condacts are hybrids that match *and* modify WM. Messages pass in one direction for conditions and actions but in both directions for condacts. Procedural knowledge is encoded via conditions

```
CONDITIONAL ConceptWeight
   Conditions: (Object state:s object:o)
   Condacts: (Concept object:o concept:c)
             (Weight object:o weight:w)
```

w\c	Walker	Table	…
[1,10>	.01w	.001w	…
[10,20>	.2-.01w	"	…
[20,50>	0	.025-.00025w	…
[50,100>	"	"	…

Fig. 2: Conditional probability of weight given concept. Only a fragment of the function is shown.

and actions. Unidirectional message passing – from WM, through conditions, on to actions, and finally back to WM – provides the forward impetus that is at the heart of the procedural use of rule memories. Declarative knowledge is encoded via condacts. Bidirectional message passing among condacts enables the kind of partial match that is at the heart of the declarative use of semantic and episodic memories.

The functions in conditionals enable encoding probability distributions, as in the fragment of semantic memory above. They also enable, for example, encoding the symbolic incompatibility knowledge used in constraint memory. These functions are multidimensional and are represented in a piecewise linear manner. There is one dimension per variable, with slices across the dimensions delimiting rectilinear regions over which a single linear function is adequate. The function in Fig. 2, for example, has two dimensions – for *weight* and *concept* – with slices occurring between concepts along one dimension and between segments of weights along the other. Each resulting region has its own linear function (in terms of just weight here).

This representation enables approximating continuous functions as closely as desired – for perceptual signal processing – but it also enables representing both discrete probability distributions and symbolic structures, through restrictions on function domains and ranges. It thus proffers a broad-spectrum *hybrid mixed* representation useable not only for this aspect of long-term memory but also for the messages at the core of the summary product algorithm. In Fig. 2, the concept is symbolic, the weight is continuous, and the value of the function is probabilistic.

The same function representation also works for working memory. Working memory is based on *predicate*s – such as Object, Concept and Weight in Fig. 2 – that are defined in terms of a name plus named-and-typed arguments. Weight, for example, has two arguments: *object*, over symbolic identifiers; and *weight*, over a segment of the continuous line. Each predicate induces a WM factor node with its own function that specifies which of its regions are present. Predicates, and thus WM functions, can combine any number of discrete and continuous dimensions, but the ranges of WM functions are limited to Boolean values. In other words, every possible element is either in working memory or not; they can't be in at some probability. This is consistent with how working memory works in Soar and with the mapping of working memory onto *evidence* at peripheral nodes in standard probabilistic graphical models [1]. However, it does differ from Soar's approach in explicitly representing – with a value of 0 – regions not present in working memory. This increases overall uniformity, but can also increase the number of regions to be processed.

Conditions, actions and condacts are specified as patterns on predicates, each of which also comprises a predicate name plus zero or more arguments. Each argument in a pattern has a name plus a value that is either a constant or a variable. In Figs. 1 and 2 all of the arguments are specified via variables (lower-case italicized symbols). Predicates can be negated to yield negated conditions and deletion actions; the action in Fig. 1, for example, is negated. Each pattern compiles into a subgraph that determines its correspondence to WM regions via messages possessing one dimension per argument. If there are constants in the pattern, an additional factor node is included to check their values. If the pattern is negated, an additional factor node is included to invert the message – positive values become 0 and 0s become 1.

Fig. 3 shows the factor graph for the conditional in Fig. 1, albeit with less important nodes omitted and the full subgraph for the `Selected` action deferred

until Fig. 4. As shown, link direction in pattern subgraphs is determined by whether they implement conditions or actions (or condacts, although not shown here). The subgraphs for all patterns within a conditional are then connected via a bidirectional join network. The resulting graph, when restricted to conditions, is similar to the combination of Rete's discrimination and join networks, including storage of intermediate match results. Rete uses *alpha* and *beta* memories to store condition matches and their combinations. In the graphical architecture, the latest message is automatically cached along each link, yielding a set of implementation-level *link memories*; where links at the end of pattern subgraphs act as alpha memories and links within the join network act as beta memories.

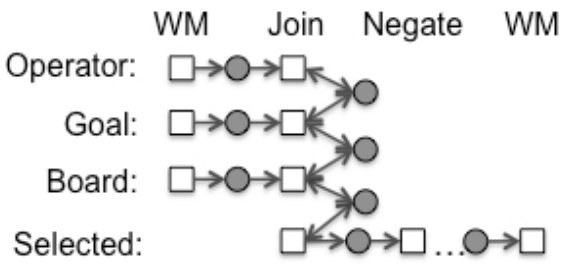

Fig. 3: Factor graph for heuristic conditional in Fig. 1. Boxes are factor nodes while circles are variable nodes.

This graphical match algorithm goes beyond Rete in efficiency by bounding the cost for condition match by the tree width rather than the number of conditions [1]. However, Rete's sharing optimizations – of tests across subgraphs within an elaboration cycle and of intermediate results across elaboration cycles – have not yet been implemented. Both of these optimizations appear feasible within the unidirectional condition subgraphs – and within those segments of the join network that only combine conditions – in a manner much like that in Rete. However, it is less clear whether this will work in bidirectional subgraphs where feedback becomes key.

The bigger gain though with the graphical approach is that the generality of the resulting mechanism yields a capability that is considerably beyond just rule match and intermediate result storage. Messages are now multidimensional continuous functions rather than partial rule matches, and they can flow not just away from working memory, but also towards it. This broadening enables a single graphical mechanism to handle conditions, actions, condacts and functions; and thus to provide a shared implementation across Soar's multiple long-term memories. As is discussed in the next section, it also yields base-level internal problem solving.

Aside from Soar's call to the decision procedure in the uppermost (decision) cycle, its three nested loops are essentially all about memory access. In the graphical memory architecture, this functionality compresses down to two nested loops: the *message cycle*, where messages pass along links in the graph; and the *graph cycle*, where message cycles repeat until quiescence and then working memory is updated. The message cycle corresponds to Soar's match cycle. The graph cycle hybridizes Soar's elaboration cycle with the elaboration-phase portion of its decision cycle.

To understand this hybridization, it is first necessary to grasp the distinction introduced in the graphical architecture between *open-world* and *closed-world* predicates, concerning whether regions not explicitly in working memory are assumed unknown or false. A region that is false – i.e., 0 – at the beginning of a graph cycle cannot become true during the cycle, increasing processing efficiency by removing

many regions from consideration; but since false can't become true without a change to working memory, chaining across such conditionals can only happen across graph cycles. Normal rules depend on closed-world predicates to keep working memory small and to implement negated conditions, implying only one cycle of rule firing per graph cycle, and thus a mapping to Soar's elaboration cycle. Semantic, episodic and constraint memory depend on open-world predicates so that values initially unknown can be determined by bidirectional processing during the graph cycle. This enables within-cycle chaining across conditionals and a full settling of the graph for access to declarative memory, indicating a mapping of the graph cycle onto Soar's elaboration phase. Muddying things even further, when rules work on open-world predicates – thus taking on a declarative aspect – it becomes possible to chain across sequences of them within a single graph cycle, again akin to Soar's elaboration phase.

The difference in chaining between closed-world and open-world predicates is implemented by chaining for a closed-world action through its WM factor node – the rightmost node in Fig. 3 – necessitating changes in working memory and a new graph cycle before results of actions in one conditional can be used in conditions of another; while chaining for an open-world predicate through the WM variable node – just to the left of the WM factor node in Fig. 3 – enabling chaining across conditionals without going through the factor node or changing working memory.

Changes to working memory occur via an action subgraph like the one shown in Fig. 4 for the Selected predicate. The top portion implements the negated action in Fig. 1, with the portion below it implementing a positive action from a different conditional. If there were additional positive actions, their subgraphs would all join at the positive-changes (+) factor node, while additional negative actions would join at the negative-changes (–) factor node. To deal with the disjunctive semantics that exists across rule actions, both of these are special function-composition factor nodes that sum their inputs rather than computing their product. As shown in the figure, a revised positive-changes message is then computed by using standard product computations to eliminate from it all regions marked for deletion in the negative message.

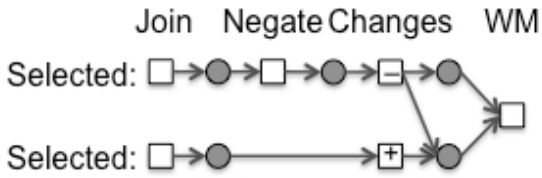

Fig. 4: Action graph for Selected predicate.

Given the aggregate positive and negative messages, the actual changes occur by altering the function stored in the WM factor node. This is a process that has much in common with learning – being an extra-graph process that modifies graph structure – although it modifies only a subset of factor node functions via a limited change-determination algorithm. Everything in the negative message is first deleted from working memory and then everything in the revised positive message is added. Closed-world modifications of working memory remain in effect until they are explicitly undone by later changes, while open-world modifications remain only as long as they are supported by conditionals in long-term memory.

Beyond the two memory distinctions already mentioned – i.e., direction of message passing and the values of unspecified regions – a third distinction has also been drawn, concerning whether variables in conditionals yield all legal values – *universal*

variables – or a distribution over the best possible value – *unique variables*. The former are essential for memories that need all exact matches, while the latter – which correspond to normal variables in probabilistic graphical models – are needed for memories that require the single best partial match. In the memory architecture, rule and constraint memory require all exact matches while semantic and episodic memory rely on distributions over the best partial match. Soar has no general distinction between universal and unique variables, but instead effectively implements universal variables in procedural memory and unique variables in both declarative memories.

The details concerning how the more general distinction is implemented within the graphical memory architecture can be found in [11]. The critical aspect for our purposes here though is that when bindings are generated during a graph cycle for an action containing a unique variable, only a single element – one with the highest value – is added to working memory (assuming there is not already one there), and all others are deleted. This is the sole locus uncovered so far where an architectural distinction is necessary between discrete and continuous arguments. If, for example, an entire region [0,3) shares the maximum value, it is necessary to distinguish whether there are three discrete alternatives competing – [0,1), [1,2) and [2,3) – or an (effectively) infinite number of continuous alternatives (which becomes a large number of ε-width segments). This is extra-graph selection code in support of changing working memory, rather than part of the summary product algorithm itself. But it is still a concrete situation in which the difference between discrete and continuous dimensions is not just in the eye of the beholder.

4 Extension to Problem Solving

The overall graphical memory capability that has just been described can be reused in service of problem solving, just as memory is reused in Soar. This means that long-term memory encodes both candidate operators and preferences among them for use in operator selection, plus operator applications that change the state and state elaborations that amplify these changes (but which can be lumped in with operator selection for the rest of this discussion). Candidate operators are added via open-world predicates so that they automatically retract when no longer valid for the state, and so that preference generation can chain on them during a single graph cycle. Preferences are generated by actions for the predefined closed-world `Selected` predicate, which includes a universal discrete numeric *state* argument and a unique symbolic *operator* argument to denote that there should be one operator per state.

There are two forms of symbolic preferences, *acceptable* and *reject*, which just amount to positive and negated `Selected` actions within functionless conditionals. The negated action in Fig. 1, for example, rejects any operator that moves a tile out of position. All regions matching an action receive values of 1 or 0, depending on whether the action is positive or negated. All relative preferences are then encoded numerically, by including functions expressing arbitrary non-negative values in conditionals that have `Selected` actions. No extensions are thus required to represent either symbolic or numeric preferences due to the multidimensional mixed nature of the function representation employed in the factor graphs.

Rather than requiring a separate preference memory, the link memories mentioned earlier automatically handle the retention of preferences. Although descriptions of Soar's memories usually omit both Rete's memories and preference memory, viewing them as implementation details, they are critical in the overall processing scheme. Here they become unified across the memory and problem solving capabilities via the generality of the graphical mechanisms originally implemented for memory; in particular, both of these varieties of implementation-level Soar memory map onto link memories at the graphical implementation level. Because preferences are maintained in link memories, they retract automatically when state changes make them invalid.

The processing of preferences for operator selection occurs via the implementation mechanisms introduced earlier for unique variables in memory. As mentioned in Section 3, Soar implements a form of unique variable in each declarative memory, but they are special purpose variants that only work there. Operator selection must instead occur via the separate decision procedure. In the graphical implementation level, these distinct aspects of Soar's use of unique variables are merged into a single more general implementation mechanism. As implied by the earlier discussion, all of the preferences get combined to yield a distribution over the operators for each region of states in the WM-change messages. The extra-graph code already in place for changing working memory then determines which operator to add for each region.

Once an operator is selected, it is applied by conditionals with closed-world actions so as to modify the state in working memory. Because of this use of closed-world predicates, only one round of operator application occurs per graph cycle, but all of the resulting changes then remain in working memory until explicitly removed. The different levels of persistence for operator selection versus operator application thus arise directly from the distinctions already existing in the memory architecture, rather than requiring additional memory distinctions in service of problem solving. Operator selection uses open-world predicates plus preferences in link memories, while operator application uses closed-world predicates. This brings a declarative aspect to operator selection – enabling openness and chaining within a single graph cycle – whether encoded in rules (with open-world actions) or in more traditional declarative forms. Operator application is purely procedural, which makes sense given that it is the core source of action and change in problem solving.

This problem solving capability has been tested in a version of the Eight Puzzle that uses continuous mental imagery to represent the board and tiles. The code consists of 18 conditionals, which compile down to a graph with 349 variable nodes, 292 factor nodes, and 718 links. The resulting graph successfully solves Eight Puzzle instances via sequences of operator selections and applications.

5 Conclusion

By exploiting the generality of the graphical implementation mechanisms previously developed in support of a broad yet theoretically elegant memory architecture, Soar-like base-level problem solving capabilities have been demonstrated. Although an architecturally defined `Selected` predicate was added in the process, the remaining functionality all grounds directly in mechanisms developed for the memory

architecture. Mechanisms reused include: *factor graphs* and *conditionals* to represent knowledge; the *summary product algorithm* to drive processing; the *mixed function representation* to represent both symbolic and numeric preferences; *within-graph link memories* to maintain generated preferences; the *open-world versus closed-world distinction* to maintain selection versus application knowledge; and the *universal versus unique variables distinction* to generate arbitrary candidate operators while selecting just the best.

A complete Soar-like problem solving capability also demands both reflection and external action, but the former is already shaping up well in separate work (while revealing unanticipated mechanism sharing with episodic memory and the nascent mental imagery capability). In general, the large amount of reuse found here augurs well as more capabilities get added towards a full implementation of a hybrid mixed variant of Soar, and as more novel architectures for autonomous general intelligence are sought that combine even broader applicability with theoretical elegance.

Acknowledgments. This effort has been sponsored by: the USC Institute for Creative Technologies; the U.S. Army Research, Development, and Engineering Command (RDECOM); and the Air Force Office of Scientific Research, Asian Office of Aerospace Research and Development (AFOSR/AOARD). 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 John Laird for helpful comments on a draft of this article.

References

1. Rosenbloom, P.S.: Rethinking Cognitive Architecture via Graphical Models. Cognitive Systems Research (In press)
2. Goertzel, B.: OpenCogPrime: A cognitive synergy based architecture for artificial general intelligence. In: 8th IEEE International Conference on Cognitive Informatics (2009)
3. Hutter, M.: Universal Artificial Intelligence: Sequential Decisions Based on Algorithmic Probability. Springer-Verlag, Berlin (2005)
4. Koller, D., Friedman, N.: Probabilistic Graphical Models: Principles and Techniques. MIT Press, Cambridge (2009)
5. Rosenbloom, P.S.: Combining Procedural and Declarative Knowledge in a Graphical Architecture. In: 10th International Conference on Cognitive Modeling (2010)
6. Laird, J.E.: Extending the Soar Cognitive Architecture. In: Artificial General Intelligence 2008: Proceedings of the First AGI Conference. Arlington, IOS Press (2008)
7. Anderson, J.R.: The Adaptive Character of Thought. Erlbaum, Hillsdale (1990)
8. Rosenbloom, P.S., Laird, J.E., Newell, A.: Meta-levels in Soar. In: Maes, P., Nardi, D. (eds.) Meta-Level Architectures and Reflection, pp. 227-240. North Holland, Amsterdam (1988)
9. Forgy, C. L.: Rete: A Fast Algorithm for the Many Pattern/Many Object Pattern Match Problem. Artificial Intelligence. 19, 17–37 (1982)
10. Kschischang, F. R., Frey, B. J., Loeliger, H.: Factor Graphs and the Sum-Product Algorithm. IEEE Transactions on Information Theory. 47, 498-519 (2001)
11. Rosenbloom, P. S.: Implementing First-Order Variables in a Graphical Cognitive Architecture. In: Biologically Inspired Cognitive Architectures 2010: Proceedings of the First Annual Meeting of the BICA Society. IOS Press, Arlington (2010)