Towards a 50 msec Cognitive Cycle in a Graphical Architecture

Paul S. Rosenbloom (Rosenbloom@USC.Edu)
Department of Computer Science and Institute for Creative Technologies
12015 Waterfront Dr., Playa Vista, CA 90094 USA

Abstract

Achieving a 50 msec cognitive cycle in any sufficiently sophisticated cognitive architecture can be a significant challenge. Here an investigation is begun into how to do this within a recently developed graphical architecture that is based on factor graphs (with the summary product algorithm) and piecewise continuous functions. Results are presented from three optimizations that leverage the structure of factor graphs to reduce the number of message cycles required per cognitive cycle.

Keywords: Cognitive architecture, cognitive cycle time, graphical models, optimization.

A common assumption underlying many cognitive architectures is that there is a core cognitive cycle that runs at ~50 msec/cycle, the time scale of the quickest human responses (once peripheral processing, such as physical movement, is subtracted out). This value is close to the 70 msec mean originally given for the Model Human Processor (Card, Moran & Newell, 1983), and matches the value used in ACT-R (Anderson, 2007), EPIC (Kieras & Meyer, 1997) and Soar (Laird, 2012). Hitting such a rate in reality is critical for architectures that are to model cognition in real time as well as for architectures that are to support construction of intelligent systems that operate on human time scales. It is less critical when the focus is purely on the non-real-time modeling of human cognition; but even there it matters in principle whether the approach can reach this time scale within neurobiological implementation constraints, as well as in practice whether the model can run fast enough for serious experimentation on complex tasks.

Driven by this constraint, there has been considerable recent work on improving the efficiency and scaling of architectural capabilities such as declarative (semantic) memory (Derbinsky, Laird & Smith, 2010; Douglass & Myers, 2010), as well as a longer history of such efforts that go back at least to work on the efficiency and scaling of procedural (rule) memory (Forgy, 1982; Doorenbos, 1993). The focus in this article is on improving the efficiency and scaling of a form of graphical model (Koller & Friedman, 2009) that is being explored as an implementation level for a broad spectrum, tightly integrated and functionally elegant *graphical cognitive architecture* (Rosenbloom, 2011a&b).

Graphical models were chosen as the basis for this architecture because of their potential for yielding a uniform approach to implementing and integrating together state-of-the-art algorithms across symbol, probability and signal processing. At their core, graphical models provide efficient computation over complex multivariate functions by decomposing them into the product of simpler subfunctions and then mapping the results onto networks of nodes and links. In a factor graph – the most general form of graphical model and the one used in the architecture – *variable nodes* represent function variables, *factor nodes* represent subfunctions, and *links* connect subfunctions with their variables (Kschischang, Frey & Loeliger, 2001). Multiple inference algorithms exist for such graphs, both exact and approximate. The graphical architecture uses a variant of the summary product algorithm (Kschischang *et al.,* 2001), a message-passing scheme that is exact for non-loopy graphs and approximate for loopy ones.

Given this algorithm, a single cognitive cycle maps onto the architecture as a *graph cycle* (GC); a solution to the graph, given evidence concerning the values of some variables, generated by passing messages until quiescence and then updating working memory (Rosenbloom, 2011c). The graph roughly corresponds to long-term memory and the evidence to working memory. A single graph cycle can include parallel waves of rule firings, access to declarative knowledge, perception, and simple forms of reasoning (including fixed chains of probabilistic reasoning, as are found for example in POMDPs). Each graph cycle is itself composed of a sequence of *message cycles* (MCs), during each of which a single message is passed along one link.

Given this core capability, the graphical architecture has already been shown to support procedural and declarative memories (Rosenbloom, 2010), plus forms of perception (Chen *et al*., 2011), imagery (Rosenbloom, 2011d) and problem solving (Chen *et al*., 2011; Rosenbloom, 2011c). However, it still operates at a time scale that is too often far above the critical 50 msec/GC threshold. Prior to the work described in this article, the average time per GC – in LispWorks 6.0.1 on a 3.4 GHz Intel Core i7 iMac with 8GB of 1333 MHz DDR3 RAM – was close to the desired value for simple tasks, such as 55 msec for the one GC involved in accessing a small semantic memory. However, the Eight Puzzle averaged 872 msec/GC when run to completion on a problem that needed 9 GCs; and a more complex virtual navigation task (Chen *et al*., 2011) – which combined perception (via a three-stage CRF), localization (via part of SLAM), and decision-theoretic choice (via a three-stage POMDP) – was even more problematic. Although this last task wasn't implemented until after some of the new optimizations described in this article were already in place, it still required 2288 msec/GC when run for 20 GCs, a factor of 46 too slow.

The obvious strategy for reducing these numbers is to decompose the problem into (1) reducing the number of

message cycles per graph cycle (MC/GC), and (2) reducing the time per message cycle (msec/MC); and then to tackle both of these subproblems individually. Across the three tasks just mentioned, the range of 55-2288 msec/GC decomposes into 564-3635 MC/GC and .1-.6 msec/MC. This article focuses on the first subproblem, exploring how to leverage the structure of the architecture's factor graphs – and the dependencies that these implicitly define – to dramatically reduce MC/GC. Work on the second subproblem – which is exploring new representations for the functions and messages at the heart of the architecture (as proposed in Rosenbloom, 2011b) – is not as far along, and is thus left as future work. We begin here with additional relevant background on the architecture's use of factor graphs and summary product, and on a set of early optimizations that were implemented prior to this work, before examining three new MC/GC optimizations.

Factor Graphs and Summary Product

In its simplest form, a factor graph embodies a variable node for each variable in the function of interest, a factor node for each subfunction in the product decomposition, and bidirectional links that connect each factor node with the variables it uses. Figure 1, for example, shows a factor graph for a polynomial function of three variables, with three variable nodes and two factor nodes. In more complex graphs, variable nodes may represent combinations of function variables, as in Figure 2, to exploit composite variables in the graph that are cross products of the function variables involved. Maintaining such cross products is crucial, for example, to solving the *binding confusion problem* (Tambe & Rosenbloom, 1994) by tracking which values of one variable are consistent with which values of another variable (Rosenbloom, 2011a).

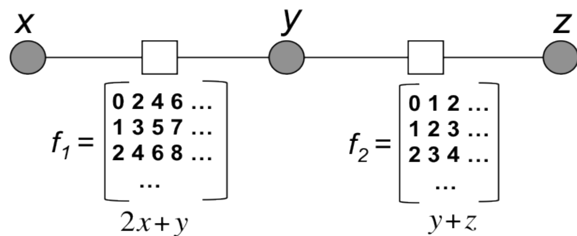

Figure 1: Factor graph for $f(x,y,z) = y^2+yz+2yx+2xz = (2x+y)(y+z) = f_1(x,y)f_2(y,z)$

By definition, factor graphs are bidirectional, so wherever there is a link between two nodes, messages pass in both directions along the link. However, in creating the graphical architecture it became clear that introducing a form of unidirectional link would enable subgraphs corresponding to the kinds of conditions and actions that occur in standard rule-based procedural memories (as in Figure 2). Conditions match to information in working memory, combining their results so that actions can then propose changes to working memory. Declarative knowledge is encoded in terms of *condacts* – which combine the effects of *cond*itions and *act*ion*s* to pass messages both to and from working memory – plus functions, such as those associated with the two factor nodes in Figure 1. Condacts yield standard bidirectional subgraphs, while conditions and actions yield subgraphs with a single active direction for message passing. This notion of link directionality is not the same as that found in Bayesian networks; the former concerns the direction of message passing, while the latter concerns how variables functionally depend on each other in factor nodes (such as in defining conditional probabilities).

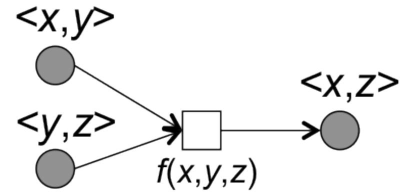

Figure 2: Factor graph for rule
`After(x,y) ∧ After(y,z) ⇒ After(x,z)`.

Conditions, actions and condacts define variablized patterns that are combined, along with functions, into *conditionals*, a generalized form of rule that forms the basic long-term-memory element in the graphical architecture. Figure 2, for example, shows a rule-like conditional for transitivity that is composed of two conditions and one action, and for which the links only pass messages towards the right. A bit of declarative memory might instead use two condacts and a function to specify the conditional probability of the first given the second, such as `Concept(x), Color(y): [(Table: Brown=.95, Silver=.05) (Dog: Brown=.7, White=.25, Silver=.05) ...]` for the classification of an object given its color. Table 1 lists basic statistics on the conditionals and their patterns for the three tasks that have been introduced. It can be seen that the Eight Puzzle is largely procedural, navigation is largely declarative, and semantic memory is more of a blend. This not surprisingly leads to a skewed distribution of link directions for the latter two, and a more balanced distribution for the first (Table 2).

Table 1: Conditional and pattern statistics for the **S**emantic Memory, **E**ight Puzzle and **N**avigation tasks.

	Conditionals	*Conditions*	*Condacts*	*Actions*
S	9	12	11	3
E	19	62	0	32
N	25	4	47	1

Table 2: Graph statistics for the **S**emantic Memory, **E**ight Puzzle and **N**avigation tasks.

	Nodes		Links	
	Factor	Variable	Uni-	Bi-
S	83	82	114	55
E	341	402	824	1
N	161	132	75	214

Each message along a link specifies a function over the variables in the link's variable node that constrains the variables' values. These functions are represented in the graphical architecture as piecewise continuous; in particular as doubly linked arrays of *n*D rectilinear (i.e., *orthotopic*) regions, where each variable maps onto a dimension, and the value function for each region is linear over its variables, as in Figure 3 (Rosenbloom, 2011b). If the function is Boolean, regions with a value of 1 are valid while regions with a value of 0 are not. If it is probabilistic, the function specifies the density over that region of variable values. However, functions can also mix these two, as in Figure 3, as well as approximate arbitrary continuous functions.

Given a new input message at a variable node, new output messages are computed for each of its links via a *pointwise product* of the new message with the incoming messages along all of the other links (except for the one along the output link). A pointwise product is like an inner product, where the value at corresponding points is multiplied; but there is no final summation over the result, so the output and input have the same rank. At a factor node, the input messages are likewise multiplied in this manner, but the factor function is also included in the product, and then all variables not in the output message are summarized out, by either integrating over them to yield marginals or maximizing over them to yield the MAP estimate. Figure 4 shows for example how evidence values of 3 for variable *x* and 2 for variable *z* propagate through the factor and variable nodes in the factor graph from Figure 1, to ultimately yield the marginal on variable *y*.

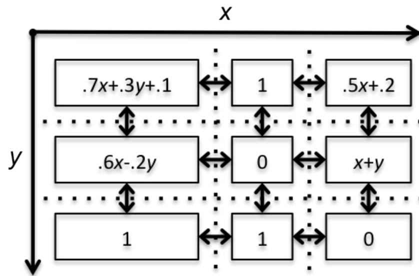

Figure 3: Piecewise continuous function as array of linear regions.

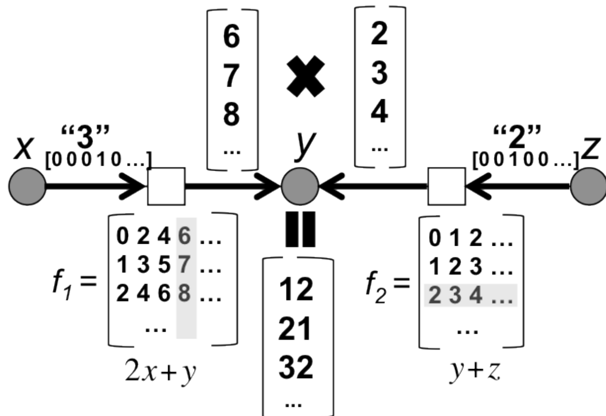

Figure 4: Computation via the summary product algorithm of the marginal on *y* from evidence on *x* and *z*.

Both product and summarization are computed in the architecture by systematically stepping through the *n*D function(s) – following the links between adjacent regions – and at each step either multiplying the corresponding regions from two functions or summarizing out a dimension of a region within one function. Summarization involves either adding the integral of the region along the dimension to the current total or computing the maximum of the current region's max and the cumulative max so far.

At the beginning of each cognitive cycle, all messages are initialized before message passing begins. If a factor node has no inputs – as is true for working memory nodes (because changes to working memory occur at decision time rather than directly via message passing) and nodes that represent functions in conditionals (which actually appear in the architectural graph in a different manner than is shown in Figures 1 and 4) – the factor node's function (once unneeded variables are summarized out), becomes the initial outgoing message. Messages from all other nodes are initialized with a value of 1, yielding no initial constraint since such messages are identities for pointwise product.

All initial messages are placed into a global message queue, which is then continuously updated as existing messages are sequentially popped and processed, and new messages are generated. The cognitive/graph cycle reaches quiescence when there are no more messages in the queue.

Preexisting Optimizations

Several optimizations that reduce MC/GC were implemented early in the development of the graphical architecture. A form of dynamic programming was incorporated that caches and reuses the last message generated along each active direction of each link. In addition, to facilitate reaching quiescence with real functions, the cached message along each link direction was updated, and a new message added to the queue, only when the difference between the old and new messages exceeded $\varepsilon = 10^{-7}$. To further reduce the number of messages to be processed, not all messages were inserted at the back of the queue. More constraining messages – ones that are 0 everywhere and thus halt all processing downstream from them, or ones that at least provide some information via values that vary over the variable's domain – were placed at the front of the queue, leaving only constant non-zero messages, which provide little discrimination, to be inserted at the back (see Figure 7a). The hope was for uninformative messages to be updated by new values along their link before being popped off the queue for processing.

Given these early optimizations, MC/GC ranged from 564 for semantic memory to 1459 for the Eight Puzzle. With a slightly enhanced queuing scheme that will be described later, the navigation task required 3635 MC/GC. For comparison purposes, this enhanced form of queuing reduced the number of messages for semantic memory by 34% (to 371) and for the Eight Puzzle by 27% (to 1062). Without these early optimizations a usable system would have been infeasible from the start, and approaching 50

msec/GC would have been impracticable. But, even with them, reaching this threshold still requires either: (1) reducing the worst-case MC/GC by 98% (from 3635 to 83), (2) reducing the worst-case msec/MC by 98% (from .6 to .014), or (3) some lesser combination of these reductions. The remainder of this article focuses on the first option.

Message Reuse Across Graphical Cycles

The early optimizations included caching of messages to enable their reuse across message cycles, but reinitialization of all messages was still required across graph cycles because there was otherwise no guarantee that modifying one message would result in all of the other messages in the graph being updated appropriately. Consider, for example, the loopy graph in Figure 5. If A is set to 0, D is set to 1, and there is no evidence concerning B and C, the graph converges to where all of the messages except the one from D are 0. If, on the next graph cycle, A becomes 1, all of the messages should settle to 1. However, without reinitialization the loop remains locked at 0. The new message to B, computed as the product of the new message from A (1) with the existing message from C (0), remains at 0, as does the new message to C. This contrasts sharply with, for example, the Rete algorithm for rule match, where messages (tokens) corresponding to unmodified regions of working memory can all be maintained and reused across cycles (Forgy, 1982).

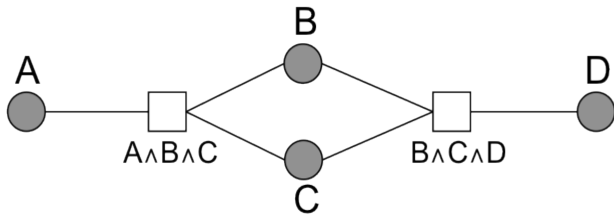

Figure 5: Loopy factor graph.

It does turn out, however, to be possible to identify segments within the overall factor graph where such reinitialization can be avoided, and where messages from the previous graph cycle can thus be reused. To do this requires preanalyzing the graph to determine which messages can possibly depend on factor node functions that may be modified between graph cycles; in particular, functions in working-memory factor nodes that are modifiable by decisions, and factor node functions specified in conditionals that are modifiable by learning. For each of these factor nodes a list of its *descendants* is first precomputed, where each descendant comprises: (1) a descendant node whose outgoing messages may be affected by messages originating at the modifiable node; and (2) a list of the descendant node's neighbors via which this influence may reach the descendant node. A message out of a node in the graph is then only reinitialized when: (1) the node is a descendent of a modifiable node that has actually been changed, and (2) the listed neighbors may pass this influence to the node so as to affect the output message.

Figure 6 shows a variant of the graph from Figure 5, but with some of the links now unidirectional, and both A and D modifiable (although shown as variable nodes, there would be a working-memory factor node feeding each). The descendants of A here are (A_F; A_V), (B; A_F, B^C^D), (C; A_F), (B^C^D; B, C), (D; B^C^D). The descendants of D are (B^C^D; D) and (B; B^C^D). If the value of A (i.e., A_V) changes, all of the messages in the graph, except for the one from D, would need to be reinitialized; but if D changes, only two messages – from D to B^C^D and from B^C^D to B – would need reinitialization. All messages not reinitialized in this fashion are retained, allowing reuse of messages that are guaranteed to remain unchanged. This optimization cannot lower the number of message cycles during the first graph cycle, but it can in all later cycles.

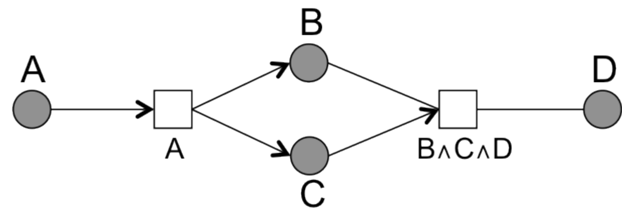

Figure 6: Variant of the loopy factor graph with a mix of bidirectional and unidirectional links.

The semantic memory test case is normally only run for a single graph cycle, but if a second GC is run without the evidence being changed, this optimization reduces the number of message cycles during the second graph cycle from 564 to 0 (saving 100%), yielding a total drop over the two graph cycles from 564 to 282 MC/GC (saving 50%). For the Eight Puzzle the number of message cycles during the second graph cycle drops from 1553 to 595 (saving 62%). Over the 9 GCs required to solve this particular problem, the average MC/GC dropped from 1459 to 1050 (saving 28%). With the navigation graph, the MC/GC drop (over 20 GCs) was from 3594 to 1359 (saving 62%).

Improved Message Ordering

The early optimizations included a heuristic for message insertion in the queue. The new approach to queuing retains the notion of constraint used there, while providing a more direct way of ensuring that messages that should be held until they contain appropriate content remain in the queue. Here we again preanalyze the graph structure, but this time to determine the *depth* of each link (in each direction). Links from nodes with no inputs – which again turn out to be working memory factor nodes plus factor nodes derived from functions in conditionals – have a depth of 0. For all other nodes not involved in loops – for which there is no unique depth – their depth is calculated as one plus the maximum of the depths of all neighbors from which they receive messages. The depth of a link in a particular direction is then simply the depth of its source node.

Message depth can then be used as a queuing heuristic that delays the processing of a message when there are

shallower ones – which thus could conceivably influence its content – also available in the queue. To implement this, the single original queue is split into a sequence of smaller queues. The first one is for *empty messages* (constant at 0) and the last one is for *full messages* (constant at 1). The former block all processing downstream from them, and can't be constrained any further. The latter are completely unconstrained, and thus not particularly useful. In between these two, a single queue was initially used for all other messages (Figure 7b), yielding the baseline results already presented for the navigation task. However, this has since been extended further, stratifying these other messages into a sequence of intermediate queues based on their depth.

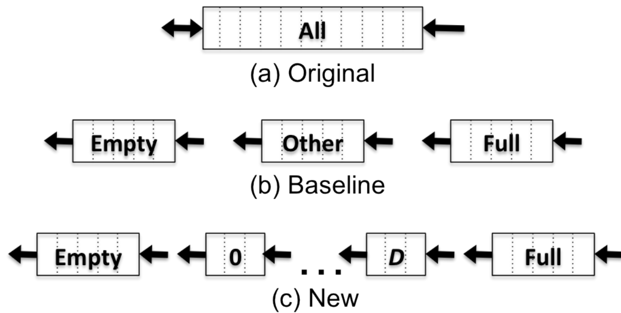

Figure 7: Three queue disciplines explored.

As shown in Figure 7c, one intermediate queue is created for each possible node depth – ordered from smallest to largest – for a total number equal to one plus the *depth of the graph*; i.e., the maximum of the depths of all of the nodes in the graph (*D*): 29 for semantic memory, 45 for the Eight Puzzle, and 76 for navigation. The last intermediate queue also handles links affected by loops. By stratifying messages in this manner, messages deeper in the graph that can be affected by shallower processing are delayed until all shallower messages are processed.

When all of the queues are included, empty messages are always sent before any other messages are considered. If there are no empty messages, then the intermediate queues are tried according to increasing depth. If there are no messages in any of these queues, the full-message queue is drained. When there are no messages in any of the queues, quiescence has been reached.

This optimization can help even during the first graph cycle, and can handle links along which messages are passed bidirectionally, as long as there are no loops. For messages affected by loops, ordering is essentially reduced to the previous baseline, with just one intermediate queue. This optimization, when enabled by itself, reduces MC/GC from the original single queue version by 60% (to 224) in semantic memory and by 43% (to 826) in the Eight Puzzle. In comparison to the three-queue baseline this is a savings of 40% for semantic memory and 22% for the Eight Puzzle. Improvement from this baseline in the navigation task lowers MC/GC by 86% (to 503).

When both this optimization and the previous one are combined, MC/GC drops by 61% (to 224) over one GC of semantic memory and by 90% (to 112) over two GCs. For the Eight Puzzle, MC/GC drops by 59% (to 602) over the 9 GCs. The gains from the three-queue baseline are 40% over one GC of semantic memory and 70% over two GCs, 43% for the Eight Puzzle, and 89% for navigation (reducing it to 391 MC/GC). Total speedup factors are thus seen that range from 2.5 to 10 across these three tasks. Concurrently, msec/MC has stayed roughly the same for semantic memory, at .1, while for the other two tasks it has dropped from .6 to .5, providing an additional speedup factor of 1.2 for these harder problems. With a new maximum of 602 MC/GC over these three tasks (for the Eight Puzzle), msec/GC would now need to be .08 – a factor of 6.25, rather than the original 46, from the current maximum of .5 – to enable all three tasks to proceed within 50 msec/GC.

(Simulated) Parallelism

Instead of reducing the number of message cycles by reducing the number of messages that need to be sent, parallelism enables multiple messages to be sent within each message cycle. One simple form of this is to send messages out in parallel along each active direction of each link of the graph, as long as there is a new message there to be sent. With such an approach, msec/GC becomes the product of msec/MC and the number of parallel message cycles (MC). In the absence of loops, MC should be bounded by the depth of the graph, again implying that the structure of the graph – in particular, how messages on deeper links depend on those on shallower links – is critical. With loops, there is no obvious a priori bound.

Although the architecture has not yet been ported to parallel hardware, a message-passing discipline has been implemented that is based on a sequence of (simulated) parallel message cycles. The first optimization introduced above, of reusing messages across graph cycles, may still be relevant with parallel message cycles; however, the second is not, given that all queued messages are effectively sent during each parallel message cycle. Although this form of parallelization implies that more total messages may be sent, sending them in parallel may radically reduce MC/GC while keeping msec/MC nearly the same.

With parallel message passing turned on and no message reuse across graph cycles, the average number of messages per cycle rises to 658 for semantic memory, 2758 for the Eight Puzzle, and 3747 for navigation; yet, the average MC/GC is only 26, 33, and 76 for the three tasks. MC/GC turns out to be relatively stable within each of these tasks, with navigation running a constant 76 and the Eight Puzzle ranging from a low of 29 to a high of 36. Given a maximum of 76 MC/GC across these three tasks, 50 msec/GC becomes feasible with an msec/MC of .7. If the communication overhead on parallel hardware is a small fraction of this, the existing maximum of .5 msec/MC should be sufficient to yield a real-time graph cycle. Such an approach also has the advantage of removing the need for a global queue, enabling message passing to be truly local.

When (simulated) parallelism is combined with message reuse across graph cycles, the average number of messages

per GC remains at 658 for semantic memory, but drops to 1962 for the Eight Puzzle and 3637 for navigation, yielding reductions of 29% and 3% for these latter two. The average MC/GC becomes 26, 28, and 76 for the three tasks, a 15% gain for the Eight Puzzle but no change for the other two.

Conclusion

With serial message passing, the first two optimizations introduced here reduce MC/GC across semantic memory, the Eight Puzzle and a navigation task by a factor of 2.5-10. Given that the optimizations also reduced the time per message cycle for the harder problems by a factor of 1.2, the total gain in time per cognitive cycle is a factor of 3-12. When considering the worst case over these three tasks, an additional factor of 6.25 is now needed to achieve 50 msec per cognitive cycle, a significant improvement over the factor of 46 that was needed at the start.

Parallelization provides a somewhat different approach to reducing MC/GC, by sending messages in parallel within message cycles. If close to the full amount of potential parallelism can be achieved on parallel hardware, it provides a path, albeit a more costly one in terms of hardware, for immediately reaching the 50 msec threshold. Even on a workstation with 2-8 cores, it may be able to help significantly in reaching this threshold, particularly if some form of the message ordering optimization were able to eliminate messages that don't really need to be sent within early message cycles (which tend to be the most computationally intensive).

For the future, it will be important to explore whether message reuse across graph cycles can be extended to a larger fraction of the graph, whether there is an analogue of the node-depth optimization that works for loopy graphs, and what would happen with a deployment on true parallel hardware. It is also important to investigate what additional gains may be had in terms of msec/MC, where a sparse function representation is currently being explored, but where other possibilities also exist. It may also ultimately prove worthwhile to consider switching from summary product to algorithms that are more approximate, based on sampling, particle filters, or variational methods. This may become particularly critical as the task complexity continues to scale up in various ways.

Acknowledgements

This work has been sponsored by the U.S. Army. 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.

References

Anderson, J. R. (2007). *How Can the Human Mind Occur in the Physical Universe?* Oxford: Oxford University Press.

Card, S. K., Moran, T.P. & Newell, A. (1983), *The Psychology of Human-Computer Interaction*. Hillsdale, NJ: Lawrence Erlbaum Associates.

Chen, J., Demski, A., Han, T., Morency, L-P., Pynadath, P., Rafidi, N. & Rosenbloom, P. S. (2011). Fusing symbolic and decision-theoretic problem solving + perception in a graphical cognitive architecture. *Proceedings of the 2nd International Conference on Biologically Inspired Cognitive Architectures*.

Derbinsky, N., Laird, J. E. & Smith, B. (2010). Towards efficiently supporting large symbolic declarative memories. *Proceedings of the 10th International Conference on Cognitive Modeling*.

Doorenbos, R. B. (1993). Matching 100,000 rules. *Proceedings of the 11th National Conference on Artificial Intelligence*.

Douglass, S. A. & Myers, C. W. (2010). Concurrent knowledge activation calculation in large declarative memories. *Proceedings of the 10th International Conference on Cognitive Modeling*.

Forgy, C. L. (1982). Rete: A Fast Algorithm for the Many Pattern/Many Object Pattern Match Problem. *Artificial Intelligence*, *19*, 17-37.

Kieras, D. E. & Meyer, D. E. (1997). An overview of the EPIC architecture for cognition and performance with application to human-computer interaction. *Human-Computer Interaction*, *12*, 391-438.

Koller, D. and Friedman, N. (2009). *Probabilistic Graphical Models: Principles and Techniques*. Cambridge, MA: MIT Press.

Kschischang, F. R., Frey, B. J. & Loeliger, H. (2001). Factor graphs and the sum-product algorithm. *IEEE Transactions on Information Theory*, *47*, 498-519.

Laird, J. E. (2012). *The Soar Cognitive Architecture*. Cambridge, MA: MIT Press. In press.

Rosenbloom, P. S. (2010). Combining procedural and declarative knowledge in a graphical architecture. In *Proceedings of the 10th International Conference on Cognitive Modeling*.

Rosenbloom, P. S. (2011a). Rethinking cognitive architecture via graphical models. *Cognitive Systems Research*, 12, 198-209.

Rosenbloom, P. S. (2011b). Bridging dichotomies in cognitive architectures for virtual humans. *Proceedings of the AAAI Fall Symposium on Advances in Cognitive Systems*.

Rosenbloom, P. S. (2011c). From memory to problem solving: Mechanism reuse in a graphical cognitive architecture. *Proceedings of the Fourth Conference on Artificial General Intelligence*.

Rosenbloom, P. S. (2011d). Mental imagery in a graphical cognitive architecture. *Proceedings of the Second International Conference on Biologically Inspired Cognitive Architectures*.

Tambe, M. & Rosenbloom, P. S. (1994). Investigating production system representations for non-combinatorial match. *Artificial Intelligence*, *68*, 155-199.