*** To be published in *Biologically Inspired Cognitive Architectures* ***

Efficient Message Computation in Sigma's Graphical Architecture

Paul S. Rosenbloom[1,2], Abram Demski[1,2] and Volkan Ustun[1]
[1] *Institute for Creative Technologies, University of Southern California, 12015 Waterfront Dr., Playa Vista, CA, 90094, USA*
[2] *Department of Computer Science, University of Southern California, 941 W. 37th Pl., Los Angeles, CA, 90089, USA.*
rosenbloom@usc.edu, ademski@ict.usc.edu, ustun@ict.usc.edu

Abstract
Human cognition runs at ~50 msec per cognitive cycle, implying that any biologically inspired cognitive architecture that strives for real-time performance needs to be able to run at this speed. Sigma is a cognitive architecture built upon graphical models – a broadly applicable state-of-the-art formalism for implementing cognitive capabilities – that are solved via message passing (with complex messages based on *n*-dimensional piecewise-linear functions). Earlier work explored optimizations to Sigma that reduced by an order of magnitude the number of messages sent per cycle. Here, optimizations are introduced that reduce by an order of magnitude the average time required per message sent.

Keywords: Cognitive architecture, graphical models, Sigma, efficiency, message passing

1 Introduction

Sigma is a *cognitive architecture* – an integrated computational model of the fixed structure underlying intelligent behavior – whose development is driven by a trio of desiderata: (1) *grand unification*, spanning not only traditional cognitive capabilities but also key sub-cognitive aspects such as perception, motor control, and affect; (2) *functional elegance*, yielding broad cognitive (and sub-cognitive) functionality – ultimately all that is necessary for human(-level) intelligence – from a simple and theoretically elegant base; and (3) *sufficient efficiency*, executing quickly enough for large-scale experiments in modeling human cognition or for real-time applications involving virtual humans, intelligent agents or robots (Rosenbloom, 2013).

Achieving these desiderata in Sigma is critically dependent on its *graphical architecture*, a layer based on *graphical models* (Koller & Friedman, 2009) that sits beneath the cognitive architecture and which implements it. The graphical architecture supports grand unification through a *mixed* (symbolic + probabilistic) *hybrid* (discrete + continuous) language; functional elegance via a small but general set of mechanisms; and sufficient efficiency through the range of state-of-the-art algorithms that are reducible to the *summary-product* message-passing algorithm on graphical models (Kschischang, Frey, & Loeliger, 2001). Sigma's graphical architecture bears a family resemblance to neural architectures, with graphical models providing an inherently parallel computational model in which processing happens locally at nodes in the graph. Some forms of neural networks in fact map directly onto graphical models (Jordan & Sejnowski, 2001); however, graphical models have the advantage of providing a particularly simple path for implementing symbolic processing and for integrating it with sub-symbolic processing.

Most of the published results to date from Sigma concern grand unification and functional elegance, focusing on new capabilities in areas such as learning (Rosenbloom, Demski, Han, & Ustun, 2013), memory (Rosenbloom, 2010), decision-making & problem solving (Rosenbloom, 2011b) (Chen, et al., 2011), and perception (Chen, et al., 2011), and how they are produced from a relatively small combination of core mechanisms. Yet sufficient efficiency can also be critical in models of both natural and artificial intelligence. There is the general concern of simply having models that run quickly enough for intended experiments and applications. But, when the goal is either real-time cognitive models or human-like applications – such as virtual humans for education and training (Swartout, et al., 2014) – the ability to match the human cognitive cycle time of ~50 msec/decision also becomes a central concern.

The predominant computational process within Sigma's cognitive/decision cycle is the application of the summary-product algorithm to compute results from graphical models; that is, the passing of messages until quiescence is reached. In consequence, a simple way to decompose time per decision in Sigma is into: (1) the number of messages sent; and (2) the average time per message sent. The total time per decision is then the product of these two factors.[1] Previous work on reaching the 50 msec/decision threshold in Sigma focused on the first factor, yielding an overall improvement factor with a single processing core of between 3 and 12 across a range of sample problems (Rosenbloom, 2012). However, the most difficult problem – a POMDP – still required over 300 msec per decision, and more complex problems have since been implemented in Sigma. Simulated parallelism was also investigated in this earlier work, but true parallelism was not implemented. In this article, the focus shifts to reducing the second multiplicative factor, the average time per message sent.

Although the operations performed by the summary-product algorithm – pointwise products of pairs of messages plus summarizing out variables from individual messages via sum/integral or max – are simple in principle, the complexity of Sigma's messages can make their efficient implementation challenging. To support a mixed hybrid language, messages in Sigma are uniformly represented as *n-dimensional piecewise-linear functions* (Figure 1). Each function has one dimension per variable in the message, with an outer product among them enabling tracking of consistent bindings across the variables. The resulting *n*-dimensional space is decomposed into an array of regions, with each region embodying a linear function (which can reduce to a constant in special cases). Messages in Sigma may be as small as zero regions, when there are no variables/dimensions, or as large as thousands or millions of regions across multiple variables/dimensions.

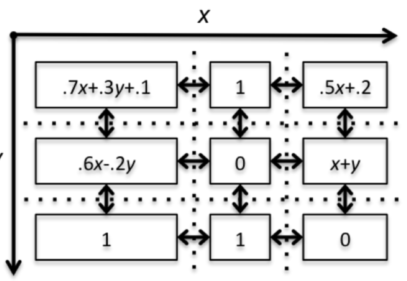

Figure 1: 2D piecewise-linear function

Messages in Sigma are thus large, structured entities. This contrasts with the typical single-value messages implicit in most neural networks, although there is a similarity to the tensors used in (Smolensky, 1990). Messages in graphical models are most typically structured as simple tables (for discrete functions) or mixtures of Gaussians (for continuous

[1] Each cycle also involves initialization, decisions, and learning, but message passing is the dominant cost at present.

functions), but here too there is work on more complex forms. For example, classic work on context-specific independence in Bayesian networks led to decomposing messages via tree-structured functions (Boutillier, Friedman, Goldszmidt, & Koller, 1996); and more recent work with cluster graphs explores several forms of structured message passing (Gogate & Domingos, 2013).

The structured nature and potentially large size of Sigma's messages demands extra attention to aspects of the efficiency of the summary-product algorithm not normally the focus. In this article, optimizations for reducing the time to compute individual messages are explored within four general categories: (1) reducing the number of times a message is (re)computed before it is sent; (2) reducing the number of operations required for computing messages; (3) reducing the number of regions that must be processed during an operation; and (4) reducing the cost of individual operations. The benefits of these different categories of optimizations interact multiplicatively, so a large gain is potentially available from their combination.

Following a bit more necessary background on Sigma, the optimizations that fall within each of these four categories are explored in the body of this article. Although the optimizations are all presented in the context of Sigma, their broader applicability is briefly discussed as part of the summary at the end of the article.

2 Sigma

Sigma's cognitive architecture is based on *predicates*, *conditionals* and *functions*. Predicates enable the representation of relational knowledge, with each predicate specified in terms of a name and a set of typed arguments. The argument types may vary in extent and may be discrete – either symbolic or numeric – or continuous. For example, a concept can be represented as a predicate with one symbolic argument: `concept(value:{walker table dog human})`; and the state of the board in the Eight Puzzle can be represented as a predicate with one discrete argument (for the *tile*) and two continuous arguments (for *x* and *y*): `board(x,y:[0-3) tile:[0:8])`. Each predicate defines its own segment of Sigma's *working memory* (WM) and *perceptual buffer* (PB).

Conditionals enable the representation of generalized conditional knowledge via a deep blending of concepts from rule-based systems and probabilistic networks. Each conditional is specified in terms of a set of predicate patterns, and possibly a function (Figure 2). Patterns may act as *conditions* or *actions*, as in rules. However, they may also act as *condacts*, a composite bidirectional form that supports the conditionality found in probabilistic networks when combined with functions that represent probability distributions. Conditionals structure Sigma's *long-term memory* (LTM).

Functions encode relationships among variables, such as joint or conditional probability distributions, although the dimensions of the functions may in general be continuous, discrete or symbolic, and the values of the functions may be arbitrary non-negative numbers (which can be limited to [0,1] for probabilities and to 0 (*false*) and 1 (*true*) for symbols). Sigma's working memory and perceptual buffer each consists of a set of such functions. Long-term memory may also include such functions, along with its conditionals. Prior to Sigma version 33 (released March 3, 2014), LTM functions were always specified as optional components of conditionals, as in Figure 2. Now, however, they can also be specified as optional components of predicates. The expectation is that this latter usage will ultimately replace most, if not all, uses of the former, but at least for now both are allowed.

```
CONDITIONAL Object-Location-Map
   Conditions: Object(value:o)
   Condacts:   Location(x:x)
   Function(x,o): .25
```

Figure 2: Map conditional for initial uniform probability of objects given locations in a discrete 1D corridor (for use in SLAM; i.e., *Simultaneous Localization and Mapping*).

Sigma's graphical architecture sits below its cognitive architecture, and serves to implement it. The graphical architecture is based on *factor graphs* and the *summary-product algorithm* (Kschischang, Frey, & Loeliger, 2001), plus a function/message representation based on *n-dimensional piecewise-linear functions* (Rosenbloom, 2011a). Factor graphs are a general form of undirected graphical model composed of variable and factor nodes. Factor nodes embody functions – including all of those defined in the cognitive architecture plus others only relevant within the graphical architecture – and are linked to the variable nodes with which they share variables. In its simplest form, there would only be one variable per variable node, but Sigma supports variable nodes that represent sets of variables. A factor graph (Figure 3) implicitly represents the function defined by multiplying together the functions in its factor nodes. Or, equivalently, a factor graph decomposes a single complex multivariate function into a product of simpler factors.

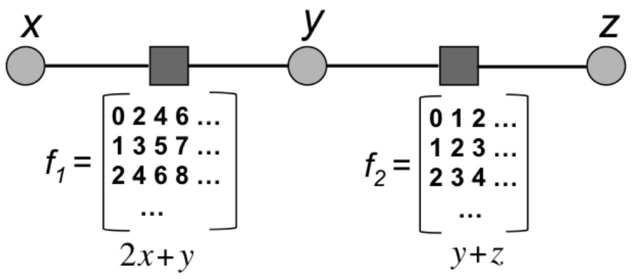
x
y
z
f1 = [0 2 4 6 … ; 1 3 5 7 … ; 2 4 6 8 … ; …]
f2 = [0 1 2 … ; 1 2 3 … ; 2 3 4 … ; …]
2x+y
y+z

Figure 3: Factor graph for the algebraic function $f(x,y,z) = y^2+yz+2yx+2xz = (2x+y)(y+z) = f_1(x,y)f_2(y,z)$

The summary product algorithm computes messages at nodes and passes them along links to neighboring nodes. A message along a link represents a function over the variables in the link's variable node. Given that a variable node may embody multiple variables, functions in messages are defined in general over the cross product of their variables' domains. An output message along a link from a variable node is simply the (pointwise) product of the input messages arriving along its other links. An output message along a link from a factor node is computed by multiplying the node's function – which is itself represented in a piecewise linear manner – times the product of its incoming messages, and then summarizing out all of its variables that are not included in the target variable node, either by integrating the variables out to yield marginals or maximizing them out to yield maximum a posteriori (MAP) estimates.

An *n*-dimensional piecewise-linear function in Sigma decomposes the function's domain into *orthotopic* – *n*-D rectangular – regions (Figure 1). Each region has a linear function of its own and is doubly linked to its neighbors along all dimensions. Region boundaries are aligned into dimension-spanning *slices*, so the function as a whole implicitly forms a regular array of regions. This representation enables arbitrary continuous functions to be approximated as closely as desired via small enough regions. Discrete functions, such as probability distributions, involve decomposing domains into unit regions with constant values. Symbolic functions involve assigning labels along discrete dimensions and, in the simplest case, limiting the functional values to 0, 1. This approach is similar to how discrete digital circuits are implemented by constraining an inherently continuous substrate, although in Sigma both the discrete and continuous aspects are leveraged.

Cognitive processing in Sigma proceeds through a sequence of *cognitive/decision cycles*, each of which includes accessing long-term memory followed by decisions (via argmax) about changes to be made in working memory – including selection of operators to be applied in problem solving – and adjustment (via a form of gradient descent) of the parameters in functions. However, all of this is actually implemented down in the graphical architecture. Working memory compiles into a sector of the graphical architecture's factor graph, with conditionals compiling into more complex sectors. Functions are represented in an *n*-dimensional piecewise-linear manner and stored in factor nodes within the overall graph. Memory access in the cognitive architecture then maps onto message passing within this factor graph. As messages are sent, they are saved on links. If a new message is generated along a link, it is sent only if it is significantly different from the one already stored there.

Message passing reaches *quiescence* – and thus memory access terminates – when no new messages are available to send. Once quiescence is reached, both decisions and learning occur locally – via function modification – at the appropriate factor nodes based on the messages they have received.

3 Reducing Number of Message (Re)Computations

Before each decision, the messages in each direction along each link need to be initialized. Prior to the very first decision, most of the messages are initialized to a uniform value of 1. Prior to subsequent decisions, a dependency-based algorithm helps determine which messages need to be reinitialized back to 1 and which can instead remain as previously computed (Rosenbloom, 2012). Under a serial message-passing protocol, the message queue is then initialized with all of the messages in the graph. When a message is then removed from the queue and sent, the set of input messages to its target node is now different, leading to a computation of the output messages along all of the node's other links, and enqueuing of these new messages (possibly replacing corresponding messages in the queue that were based on earlier sets of inputs).

This incremental updating of output messages as input messages arrive is natural, but inherently inefficient. It amounts to an *eager computatio*n of messages, with message values being updated each time one of the corresponding inputs changes, even though the output message may itself not be sent for some time. The issue is that other input messages may also change in their turn before the output message is sent, with each requiring a recomputation of the message value. A *lazy computation* approach instead waits until a message is ready to be sent before computing its value. Inputs may change many times before an output message is sent, but the only combination of inputs that matters is the one in place when the output message is actually sent.

Experiments comparing eager and lazy message passing were performed in Sigma version 28 (released October 31, 2013).[2] Measurements were made for five tasks: (1) a small procedural (rule) memory task (Rosenbloom, 2010); (2) a small semantic (naïve Bayes) memory task (Rosenbloom, 2010); (3) the Eight Puzzle (Rosenbloom, 2011b); (4) a Shift-Reduce parser; and (5) the Ultimatum Game – a standard two-person game – in a controlled problem solving mode (Pynadath, Rosenbloom, Marsella, & Li, 2013). Basic data for each of these tasks, plus the comparisons of the two versions in terms of time per decision, can be found in Table 1.

Table 1: Evaluation of lazy (versus eager) message passing

Task	Conditionals	Decisions	Messages/ Decision	Eager (msec/D)	Lazy (msec/D)	% Better
Procedural	1	1	25	9	4	56
Semantic	9	1	208	65	45	31
8 Puzzle	17	7	382	121	74	39
Parsing	13	34	392	204	192	6
Ultimatum	23	93	624	196	173	12

Substantial improvements are evident, ranging from 6% up to 56%. Although this optimization is insufficient by itself to reduce msec/decision for the more complex tasks down to below 50, it does play an important role.

[2] Unless otherwise noted, results in this article are all from LispWorks on an iMac with 3.4 GHz Intel Core i7 processors. However, the results do come from multiple versions of Sigma, most often the latest release – version 33 – but occasionally, as here, from an earlier version in which the optimization was introduced and experimentation was done.

4 Reducing Number of Operations per Message

Computing a new output message requires a number of products among input messages. For variable nodes, the number of products per output message is one less than the number of messages arriving along other links. In the most general case, where every link at the node is bidirectional, this amounts to the number of links minus two. If there are *n* such links at a variable node, this implies that *n*(*n*-2) products are required to compute all of the node's output messages. However, many of these products are redundant. For example, in computing any two outputs, all but one product for each output – the product that includes the input from the other link – is potentially sharable.

One approach to implementing such sharing would be to extend message processing at nodes so as to store and reuse products whenever feasible. However, a simpler approach is available that maintains the existing node processing while *expanding individual nodes* with large numbers of bidirectional links into subgraphs that compute intermediate products and store their results, as normal, along the appropriate links. Consider, for example, the variable node in Figure 4a, which is linked to eight factor nodes. In Figure 4b, this variable node is replaced by a hierarchy of nodes that accomplishes the same end of delivering to each of its linked factor nodes the product of the messages from the seven other factor nodes (assuming the factor nodes in the hierarchy have constant functions of 1).

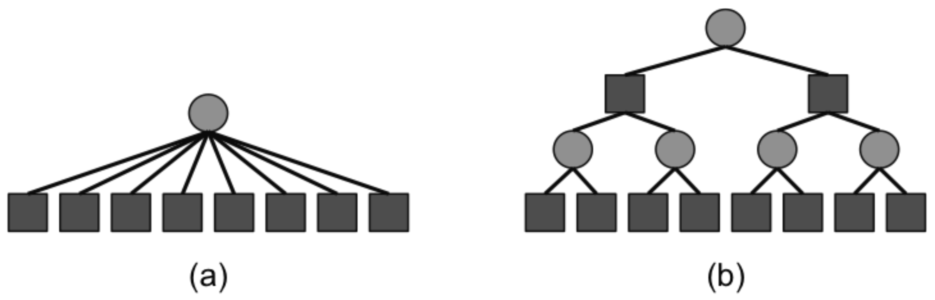

Figure 4: Expanding a variable node into a hierarchy of nodes

In such a hierarchical structure, no products are required at nodes that have fewer than three links, and each node that has three links requires three products, one for each output.[3] If we consider for simplicity the case where the number of links (*n*) connected to the original variable node is a power of two, the number of products required in the hierarchical version is $3n \sum_{0<i<\log_2 n} 2^{-i}$. For *n*=8, this is 18, which compares favorably with the 8*6 = 48 products in the original version. More critically, the number of products now grows linearly with the number of links at the original node rather than quadratically. To see this, note that extending the summation to infinity yields a total of 1, so an upper bound on the number of products is simply 3*n* (yielding an upper bound of 24 when *n* is 8). The mathematics is more complicated for arbitrary *n*, but the upper bound would still be linear.

Thus, instead of *n*-2 products per outgoing message at such variable nodes, there are at most 3. One downside of this approach is that, in expanding the graph structure, it increases the number of messages required to communicate the appropriate values among the original *n* factor nodes – the number of messages now has an upper bound of 4*n* rather than the previous fixed value of 2*n* – and thus does introduce linear overhead. For small numbers of links, the overhead can dominate, so we have set the threshold for triggering variable-node expansion at 5 bidirectional links. There has so far been no need to expand factor nodes.

[3] Technically there is one additional product at each factor node, involving its constant function of 1, but this product can be optimized out – as discussed later – and so is not considered further in the analysis here.

Of the five standard tasks used in this article, only one of them triggers this optimization – semantic memory – and it only barely does so, as the variable node for the concept has 6 bidirectional links. With this optimization, the number of messages per decision goes up from 222 to 234, but the time per decision stays roughly the same. On a larger problem of this type – a naïve Bayes classifier for word sense disambiguation (Rosenbloom, Demski, Han, & Ustun, 2013) – there is a variable node with 482 bidirectional links. The comparisons without and with the optimization show that there are 3921 versus 4663 messages/decision – an increase of 18% – but msec/decision is 2042 versus 613 (a decrease of 70%). Time per message actually goes down from .46 to .07 msec (a decrease of 85%).

5 Reducing Number of Regions Processed per Message

One key optimization of this sort that was designed into Sigma from the beginning is a form of *message minimization* that automatically removes slices, and thus automatically coalesces adjacent regions, whenever all pairs of corresponding regions across the slice have equivalent functions. Within the restriction that Sigma's messages must always be regular arrays of regions, this optimization can dramatically reduce the number of regions per message. Results from this optimization have not previously been published, so results are shown in Table 2 for time (msec/decision), and the maximum and average number of regions in a message. From these results, it can be seen that the average is lower with message minimization across all of the tasks, and the maximum is lower for the two most complex ones. For these last two tasks, substantial improvements in time per decision are also seen, of 97% and 74%.

Table 2: Evaluation of message minimization

	No Minimization			**Minimization**		
Task	**msec/D**	**Max**	**Average**	**msec/D**	**Max**	**Average**
Procedural	1	12	12	1	12	10
Semantic	8	36	8	10	36	7
8 Puzzle	38	162	45	38	162	27
Parsing	3355	17000	311	96	8464	87
Ultimatum	666	6000	251	174	625	27

A second optimization of this class has been introduced via *automated condition ordering* during the compilation of Sigma's conditionals. This automatically determines the order in which messages from conditions are joined in the graph, and thus how many variables – and thus dimensions (and regions) – must be maintained in each message as the subgraphs are joined. The approach is similar to the greedy condition ordering algorithm Soar uses in support of efficient Rete match (Forgy, 1982) – which itself relates to *join ordering* in databases – but there are two key differences. First, as Soar joins successive conditions, the number of variables implicitly represented in its tokens increases monotonically, whereas, in Sigma, variables are summarized out as soon as they are no longer needed in later patterns. Second, because Sigma's variables are typed, it is possible to use these types heuristically in determining the likelihood of large messages involving them.

During compilation of a conditional in Sigma, the choice of which condition should come next is based on a heuristic estimate of the *message volume* that would result after the condition is joined with the previous ones in the sequence; computed as the product of the spans of the message's dimensions. The span of a symbolic dimension is assumed to be the number of symbols along it, the span of a discrete dimension is assumed to be 10 and the span of a continuous dimension is assumed to be 100. These values are obviously approximate and heuristic, but have worked fine to date.

Suppose, for example, there are three conditions – `A(w, x, y), B(y, z) and C(x, z)` – where `w` and `x` are symbolic with 120 and 2 possible values respectively, `y` is discrete numeric, and `z` is continuous. If we look at just the spans of the conditions themselves, we get 2400=120*2*10, 1000=10*100 and 200=2*100, which would suggest starting with condition `C`; however, `w` would be summarized out after condition `A` because it is not needed in either conditions `B` or `C`, implying that the span we actually care about for it is just 20=2*10 (maintaining `x` and `y`), so condition `A` is preferred. The spans after the condition sequences `AB` and `AC` are now 200=2*100 (maintaining `x` and `z`) and 1000=10*100 (maintaining `y` and `z`), so condition `B` is preferred next, with condition `C` last.

Experimental comparisons are tricky here, because they depend on the quality of the initial manual ordering used, so we will be satisfied with just showing the potential of this technique by comparing a version of the shift-reduce parser (which is similar, but not quite identical, to the version used in the rest of this paper) that (1) has a known bad ordering, to the same program that is (2) automatically reordered, and (3) ordered manually. Although, as with variable-node expansion, this optimization alters the structure of the factor graph, it should not change the number of messages sent.

Table 3 shows these results from Sigma version 34 (unreleased), which is the first version to have the complete reordering algorithm implemented as described here (earlier versions used an approach that was closer to minimizing the volume of the messages from the individual conditions rather than minimizing the volume after they are joined with previous conditions). The results here are indistinguishable between manual and automatic reordering. The maximum and average number of regions improve over the bad ordering by factors of 10 and 11, respectively, and time improves by a factor of 98.

Table 3: Evaluation of automatic condition reordering

Ordering	msec/D	Max Regions	Avg. Regions
Bad	1466	86700	686
Automatic	15	8464	62
Manual	15	8464	62

A third optimization in this class takes advantage of *sparsity* in an input to a product. The standard implementation of message product in Sigma first initializes the output message by creating a set of regions based on the union of the slices in the two input messages. It then traverses the three messages in synchrony, computing the function to assign to each region of the output message by multiplying the functions in the corresponding input regions, and then minimizing the resulting message. However, if one of the input messages is sparse, in the sense that its value is 0 nearly everywhere, while the other input message has many regions, then many output regions will be created before it is discovered that most will be 0 and thus can be coalesced back down to a few regions. A canonical example is where one message acts as an index – with a 1 in one location and 0s everywhere else – for accessing one element in a large vector of values represented by the other message (Figure 5). The result message should have only one non-zero region, corresponding to the non-zero location in the index message, with 0s everywhere else. At worst, this should include only three regions, two of which are 0, but many regions must first be created and eliminated before this reduced output is reached.

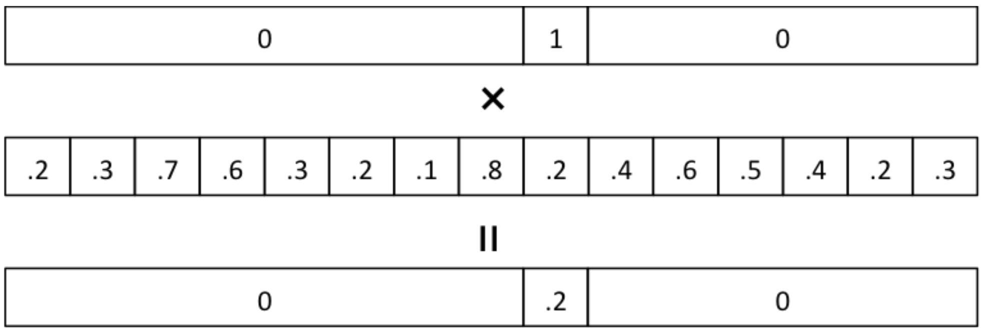

Figure 5: Using pointwise product to index a large vector

To avoid this unnecessary expense, if the first message is sparse – that is, the proportion of its volume that is non-zero is small enough – an output message is initialized based on all of the slices from the first message plus only those slices from the second message that (1) differ from these and (2) fall within non-zero regions of the first message. This approach is actually effective enough that it is worthwhile to set the sparse threshold quite high, at .85. The results of experiments with and without this optimization are shown in Table 4. For all but the simplest program, significant savings result, ranging from 9% to 48%.

Table 4: Evaluation of sparse message product (msec/D)

Task	Normal	Sparse	% Better
Procedural	1	1	0
Semantic	11	10	9
8 Puzzle	50	38	24
Parsing	184	96	48
Ultimatum	264	174	34

Taken even further, leveraging sparsity can lead to an entirely different representation for *n*-dimensional piecewise-linear functions. The general idea is to define a default value for each message, and then only explicitly represent those regions that have functions that differ from this default. The non-default regions are indexed along each of their dimensions according to their projections onto the dimensions' axes. Detecting region overlap here becomes much like detecting object collision via *axis-aligned bounding boxes* within computer graphics (Jiménez, Thomas, & Torras, 2001), except that separate indices are created along each dimension, as opposed to a single bounding volume hierarchy for the entire *n* dimensional structure. The dimensional indices were originally just ordered lists, but are now *range trees* (Bentley, 1979) for improved accessing efficiency.

This approach is still limited to regions that are orthotopic, but there need not be a neat array of regions (Figure 6), and there is a possibility of extending the approach to more complex shapes, such as *convex polytopes* (*n*-D polygons).

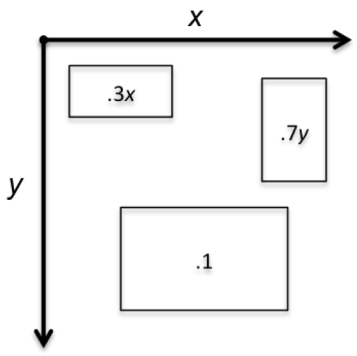

Figure 6: Sparse function representation with unspecified areas defaulting to 0

What this sparse representation accomplishes is to reduce the number of regions that need to be explicitly processed by: (1) omitting default-valued regions (while adding distinct default processing); and (2) coalescing any pair of adjacent regions that have the same function as long as their combination yields an orthotope. As with the expansion of variable nodes into subgraphs, this approach does introduce additional overhead, due to the more general form of region-overlap detection that is involved, but this can be more than counterbalanced when the number of regions to be processed is sufficiently reduced.

The full sparse representation is not yet integrated into the current version of Sigma; it instead runs in Sigma version 12 (released October 12, 2012). Experimental results comparing the normal and sparse representations in this version of Sigma can be found in Table 5. Because this is an older version of Sigma, not all of the same programs – or the same versions of similar programs – run in it. In particular, the Ultimatum game does not run, and the versions of the Eight Puzzle and of Shift-Reduce Parsing are different. There are also small differences in number of messages between the normal and sparse versions, but not large enough to be significant.

Table 5: Evaluation of the sparse representation

	Normal			**Sparse**			**% Better**
	msec/D	**Max Regions**	**Avg. Regions**	**msec/D**	**Max Regions**	**Avg. Regions**	**msec/D**
Procedural	3	12	9.5	1.7	2	1.6	43
Semantic	24.2	36	5.7	32.8	10	2.4	-26
8 Puzzle	383.8	864	37.4	94.5	27	1.5	75
Parsing	602.5	8464	92.7	286.1	2370	12	53

The results here are promising, but still preliminary. All of the programs experience a significant reduction in average number of regions, with semantic memory witnessing a 58% improvement and the other three experiencing improvements of between 83% and 96%. The reduction in regions is insufficient to overcome the overhead for semantic memory, so a slow down is seen there, but the other three programs show efficiency improvements in the range of 43% to 75%. If the mixed results seen here remain upon more thorough experimentation with the latest version of Sigma, selective integration of sparse representations may make sense.

6 Reducing Cost of Individual Operations

Two particular optimizations are discussed in this section, with the first involving a variant representation for *n*-dimensional piecewise-linear functions. So far in this article, two different representations of such functions have been introduced, one based on a doubly linked list of regions, and the other based on an indexed set of non-default regions. A third – an *array of regions* – has also been incorporated into Sigma in a limited manner. Instead of replacing either of the other representations in messages, it is presently only used locally within individual operations that would normally process doubly linked lists of regions – incoming messages are translated into the array format, processed in it, and then translated back into the doubly linked representation. This approach is particularly useful when the messages being represented are not sparse yet the operations on them require the equivalent of random access to locations. To date, this has primarily found use in initializing and copying messages.

The time per decision for the five standard tasks without either of the optimizations in this section can be found in the *Neither* column of Table 6, while the time per decision with arrays of regions, and the comparison of it to the *Neither* column, can be found in the *Region Arrays* columns. For the simplest task there was no measurable improvement, but on the other four tasks the improvement ranged from 13% to 41%.[4]

Table 6: Evaluation of individual operation optimizations

	Neither	Region Arrays		Special Case		Both	
Task	**msec/D**	**msec/D**	**% Better**	**msec/D**	**% Better**	**msec/D**	**% Better**
Procedural	1	1	0	1	0	1	0
Semantic	16	12	25	12	25	10	37
8 Puzzle	62	54	13	45	27	38	39
Parsing	229	172	25	155	32	96	58
Ultimatum	370	220	41	334	10	174	53

The fact that this approach is cost effective even with the need to translate into and out of the array representation within each operation has led to the not-yet-implemented idea of entirely replacing the doubly linked representation with this array representation; or, even better, replacing it with a hybrid between the array and sparse representations. The doubly linked approach was originally chosen to facilitate changing portions of messages in place, not for easy access to arbitrary regions. However, it turns out that altering functions in place is rarely called for, as new copies of messages are almost always required whenever changes are made. Thus the gains the representation promised appear to be more than offset by the costs it incurs.

The other approach to reducing the cost of individual operations that has seen considerable recent attention is *special case optimization*. At the message level, for example, when the entire message is a constant 0, the product will be 0 no matter what the other message is, and when a message is a constant 1, products with it can simply be skipped (as was mentioned earlier). Similarly, at the region level a constant function of 0 or 1 can similarly bypass the normal product of region functions. More such optimizations have also been incorporated, but a detailed list of them would not be terribly interesting. The time per decision for the five standard tasks with this optimization, and the comparison of it to the *Neither* column, can be found in the *Special Case* columns of Table 6. For the simplest task there was again no measurable improvement, but on the other four tasks the improvement ranged from 10% to 32%.

Combining both optimizations – in the *Both* columns of Table 6 – reveals that their effects are roughly additive, and since the task that improved the most with the first optimization improved the least with the second, the aggregate improvement – ignoring the simplest task where there was no improvement – ranged from 37% to 58%. So, together, these two simple optimizations roughly double Sigma's speed on problems of any complexity.

7 Summary

To recap the total improvement achieved by the optimizations discussed in this article, summary numbers can be found in Table 7 that compare Sigma version 24 (released June 25, 2013) – the version before the introduction of any of the optimizations discussed here (other than message minimization) – versus Sigma version 34, by when all but the full sparse message representation were incorporated. However, it should be noted that these optimizations were not the only changes made to Sigma during this time period. The most significant additional optimization reduced the number of

[4] Not included in these results are the much bigger gains yielded by this optimization for the time to initialize large LTM functions before the first decision.

messages sent by incorporating Rete-like sharing of graph structures across similar conditions (Forgy, 1982). To compensate for this potential confound, overall improvement factors are shown in the table for both msec/decision (the standard measure used in this article) and msec/message (which helps factor out any improvements in number of messages).

Table 7: Summary comparison

	Sigma 24		Sigma 34		Improvement Factor	
Task	M/D	msec/D	M/D	msec/D	msec/D	msec/M
Procedural	23	4	25	1	4	4
Semantic	212	32	234	9	4	4
8 Puzzle	481	367	335	34	11	8
Parsing	464	1060	382	61	17	14
Ultimatum[5]	718	516	602	164	3	3

The table shows overall improvements ranging from a factor of 3 up to 17, with time per message improving by factors of 3 to 14. Only the two easiest tasks ran in less than 50 msec per decision in Sigma version 24, with the other three too slow by factors ranging from 7 to 21. The Eight Puzzle now also runs within this time limit, and parsing comes very close to it. The remaining task is only a factor of 3 too slow at this point. The multiplicative combination of the earlier reductions in the number of messages sent times the reductions here in the time per message sent imply that Sigma's overall time per decision has been improved by one to two orders of magnitude.

Further optimizations are anticipated that should get Sigma even closer to the goal of pervasively running at <50 msec/decision. Two already mentioned are the sparse and region-array representations. Taking advantage of potential parallelism could also make a dramatic difference in time per decision, and thus in time per message sent (Rosenbloom, 2012). One additional key optimization still to come is a form of *incremental message passing* that would further reduce the number of regions processed per message by operating only on regions that have changed. At present, a message is sent along a link, and thus processed, whenever it differs from the previous message along the link; however, even if the difference is in only one region, the entire message is (re)processed. With incremental message processing, only regions that change would need to be (re)processed.

Such additional optimizations will become even more critical as we push towards tasks that require millions of regions per message, such as in current work that explores the use of distributed vector representations in Sigma (Ustun, Rosenbloom, Sagae, & Demski, 2014). However, none of these optimizations can yield hard real-time guarantees. If the system is ultimately fast enough, this simply may not be an issue, as it is no longer an issue for Soar. But, if not, further modifications may still be needed to ensure satisfying hard real-time constraints.

Although the optimizations presented in this article have all been explored in the context of Sigma, their potential applicability is broader. Special case optimizations are of course already widely used across many forms of software. Lazy message passing should be useful in any form of graphical model as long as the solution method is a form of message passing. Expanding variable nodes should be useful whenever message passing is used in a highly connected region of an undirected graph. Whenever complex messages are used, choosing an appropriate message representation, leveraging message sparsity, and keeping messages minimized can all be key issues. Automated condition ordering may be critical in any architecture where the joining of variablized patterns is a common inner-loop operation.

[5] The versions of the Ultimatum game that run in Sigma 24 and Sigma 34 differ, so the numbers here are illustrative but not completely comparable.

8 Acknowledgements

This work was sponsored by the U.S. Army, the Air Force Office of Scientific Research, and the Office of Naval Research. Statements and opinions expressed may not reflect the position or policy of the United States Government, and no official endorsement should be inferred.

References

Bentley, J. L. (1979). Decomposable searching problems. *Information Processing Letters , 8*, 244-251.

Boutilier, C., Friedman, N., Goldszmidt, M., & Keller, D. (1996). Context-specific independence in Bayesian networks. *12th Conference on Uncertainty in Artificial Intelligence*, (pp. 115-123).

Boutillier, C., Friedman, N., Goldszmidt, M., & Koller, D. (1996). Context-specific independence in Bayesian networks. *Proceedings of the 12th Conference on Uncertainty in Artificial Intelligence*, (pp. 115-123).

Chen, J., Demski, A., Han, T., Morency, L.-P., Pynadath, D., Rafidi, N., et al. (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*, (pp. 64-72).

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

Gogate, V., & Domingos, P. (2013). Structured message passing. *Proceedings of the 29th Conference on Uncertainty in Artificial Intelligence.*

Jiménez, P., Thomas, F., & Torras, C. (2001). 3D collision detection: a survey. *Computers & Graphics , 25*, 269-285.

Jordan, M. I., & Sejnowski, T. J. (2001). *Graphical Models: Foundations of Neural Computation.* Cambridge, MA: MIT Press.

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

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

Pynadath, D. V., Rosenbloom, P. S., Marsella, S. C., & Li, L. (2013). Modeling two-player games in the Sigma graphical cognitive architecture. *Proceedings of the 6th Conference on Artificial General Intelligence*, (pp. 98-108).

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

Rosenbloom, P. S. (2010). Combining procedural and declarative knowledge in a graphical architecture. *Proceedings of the 10th International Conference on Cognitive Modeling*, (pp. 205-210).

Rosenbloom, P. S. (2011b). From memory to problem solving: Mechanism reuse in a graphical cognitive architecture. *Proceedings of the 4th Conference on Artificial General Intelligence*, (pp. 143-152).

Rosenbloom, P. S. (2013). The Sigma cognitive architecture and system. *AISB Quarterly , 136*, pp. 4-13.

Rosenbloom, P. S. (2012). Towards a 50 msec cognitive cycle in a graphical architecture. *Proceedings of the 11th International Conference on Cognitive Modeling*, (pp. 305-310).

Rosenbloom, P. S., Demski, A., Han, T., & Ustun, V. (2013). Learning via gradient descent in Sigma. *12th International Conference on Cognitive Modeling*, (pp. 35-40).

Smolensky, P. (1990). Tensor product variable binding and the representation of symbolic structures in connectionist networks. *Artificial Intelligence , 46*, 159-216.

Swartout, W., Artstein, R., Forbell, E., Foutz, S., Lane, H. C., Lange, B., et al. (2014). Virtual Humans for Learning. *AI Magazine , 34*, pp. 13-30.

Ustun, V., Rosenbloom, P. S., Sagae, K., & Demski, A. (2014). Distributed vector representations of words in the Sigma cognitive architecture. *Proceedings of the 7th Annual Conference on Artificial General Intelligence.*, (pp. 196-207).