
Article:
Burles, Nathan John orcid.org/0000-0003-3030-1675, O'Keefe, Simon orcid.org/0000-0001-5957-2474, Austin, Jim orcid.org/0000-0001-5762-8614 et al. (1 more author) (2013) ENAMeL : a language for binary correlation matrix memories : reducing the memory constraints of matrix memories. Neural Processing Letters. pp. 1-23. ISSN 1573-773X

https://doi.org/10.1007/s11063-013-9307-8

Neural Processing Letters manuscript No.
(will be inserted by the editor)

ENAMeL: A Language for Binary Correlation Matrix Memories

Reducing the Memory Constraints of Matrix Memories

Nathan Burles · Simon O'Keefe · James Austin · Stephen Hobson

Received: date / Accepted: date

Abstract Despite their relative simplicity, Correlation Matrix Memories (CMMs) are an active area of research, as they are able to be integrated into more complex architectures such as the Associative Rule Chaining Architecture (ARCA) [1]. In this architecture, CMMs are used effectively in order to reduce the time complexity of a tree search from $O(b^d)$ to $O(d)$—where b is the branching factor and d is the depth of the tree. This paper introduces the Extended Neural Associative Memory Language (ENAMeL)—a domain specific language developed to ease development of applications using correlation matrix memories (CMMs). We discuss various considerations required while developing the language, and techniques used to reduce the memory requirements of CMM-based applications. Finally we show that the memory requirements of ARCA when using the ENAMeL interpreter compare favourably to our original results [1] run in MATLAB.

Keywords correlation matrix memory · associative memory · compact representation · domain specific language · rule chaining

1 Introduction

Domain Specific Languages (DSLs) can provide many benefits to those working in a domain, by allowing solutions to be developed at the abstract level of the domain [2]. This paper presents the Extended Neural Associative Memory Language (ENAMeL), a DSL for work with binary correlation matrix memories (CMMs), designed to allow efficient applications using CMMs to be more easily developed. We show that another benefit of ENAMeL is the potential for greatly reduced memory requirements, due to the use of compact vector representations. This is important to allow CMM-based systems to scale and become able to be applied to real-world problems.

When compared to a general purpose language, previous work has often shown that the abstraction provided by a DSL can lead to greater developer productivity [2]. On the other

Nathan Burles · Simon O'Keefe · James Austin · Stephen Hobson
Advanced Computer Architectures Group
Department of Computer Science
University of York
York, YO10 5GH, UK
E-mail: {nburles,sok,austin,stephen}@cs.york.ac.uk

hand, depending on the domain, it may not be appropriate to design and implement a new DSL due to the time and costs involved.

There has been gradually more research using CMMs in recent years (for example [3] and [4]), and so the development of a DSL for this domain is worthwhile. Additionally, the existence of a DSL in this domain may help to spur further research as it lowers the requirements of entry to the field.

2 Literature Review and Related Work

A number of different approaches may be used in order to implement a DSL, with different approaches best suited to different domains. These approaches were classified by Spinellis [5], and improved upon by Mernik et al. [6]. Mernik et al.'s classifications are briefly given below:

Preprocessing A program written in the DSL is translated into constructs of a base language.

Embedding A library is written for a base language, to provide domain-specific operations to that language.

Compiler/Interpreter A DSL is implemented using standard compiler/interpreter techniques. This may involve the use of compiler writing tools, or may extend an existing compiler/interpreter, to reduce implementation effort.

Commerical off-the-shelf The application of existing tools to a specific domain, such as XML-based DSLs.

A study performed by Kosar et al. [7] concluded that while the embedded approach often requires the least initial effort for implementation, the DSL it provides is then one of the hardest to use. Conversely, implementing a compiler or interpreter requires a lot of effort, but is the easiest for end-users to rapidly write correct programs.

2.1 Advantages and Disadvantages of a DSL

A DSL can provide many benefits to the users of a domain, as programs are expressed at the same level of abstraction as the domain itself [2]. This can lead to concise, self-documenting code, that is easier to understand, validate, and modify [8]. In turn, this can help to increase productivity, reliability, and maintainability of programs [9,10]. Finally, DSLs allow domain knowledge to be conserved and reused by non-experts, providing validation and optimization at the domain level [11,12,13] as well as easy re-use of software [14].

On the other hand, DSLs also have a number of disadvantages; van Deursen et al. assert that in addition to the visible costs of designing, implementing, and maintaining a DSL, there may also be a cost of educating DSL users [2]. They also identify that there may be a loss of efficiency when compared with hand-coded software—although this assumes that the program is written by relative experts in the domain who are able to perform optimisations.

2.2 Binary Correlation Matrix Memories

A binary CMM [15] is a simple associative neural network, that has only a single matrix layer of weights. The input and output neurons are fully connected, and simple Hebbian

learning is used [16]. This provides efficient, online learning of associations, requiring only local updates to the matrix of weights.

The method used to learn associations is formalised in equation 1, where M is the CMM, x is the set of input vectors, y is the set of output vectors, and n is the number of training pairs. $\bigvee$ denotes the superposition of binary vectors or matrices, using the logical OR.

$$M = \bigvee_{i=1}^{n} x_i^T y_i \tag{1}$$

Recalling an associated vector from the CMM is likewise a simple process, shown in equation 2. A matrix multiplication is performed between the transposed input vector and the CMM. The result of this matrix multiplication is a non-binary vector, however, and so a threshold function f is applied.

$$y = f(x^T M) \tag{2}$$

A number of thresholding functions can be used during a recall operation, dependent on the application and on the data representation chosen. One of the simplest is known as Willshaw thresholding—the threshold is selected as the weight of the input vector—the number of bits contained within it that are set to 1 [15].

Willshaw thresholding can be used with any data representation in which the input vectors have a fixed weight, and guarantees that all associations learnt will be recalled accurately. If a CMM becomes saturated (with too many vectors associated in it) additional values known as "ghosts" may also be recalled. These are output bits that are incorrectly set due to unwanted interactions between pairs of vectors trained into the CMM, and may cause recall errors.

When a fixed-weight coding is used, the alternative *L-max* threshold may be applied. In this case the threshold is selected such that the l highest values in the non-binary vector are set to one, where l is the fixed weight of an output vector. This has been shown to provide improved recall performance when a noisy input vector is recalled [17].

More recently, Hobson presented a threshold function *L-wta* that can provide CMMs with an improved storage capacity over alternative thresholding methods [18]. It requires the use of fixed-weight vectors, generated using an algorithm proposed by Baum et al. [19]. This generation algorithm imposes constraints upon the vectors that are created—namely that the vector is divided into sections and that each section contains exactly one bit set to one. In the *L-wta* threshold, this constraint is used to great effect—essentially applying the *L-max* threshold within each section of the vector, with an l value of 1.

3 Extended Neural Associative Memory Language

ENAMeL has been developed to simplify the implementation of programs using CMMs. As such, there are a few basic operations that are required—the ability to create vectors and matrices, and to learn and recall associations. Research into complex CMM-based architectures such as the Cellular Associative Neural Network [20] or ARCA [1] has shown that in addition to these basic operations, a number of other operations are also necessary. These are detailed in section 4 below.

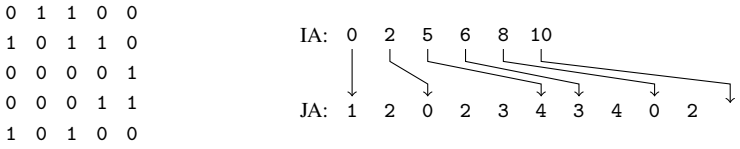

Fig. 1 An example matrix represented in explicit format (on the left) and using the binary Yale format (on the right). In the binary Yale format the first array (IA) stores the indices relating to the beginning of each matrix row in the second array (JA), which stores the column indices of bits set to one. For example, the location in JA of the set bits of the fourth row are found by accessing the 3rd and 4th locations of IA (6 and 8 respectively). The column positions of the set bits in the fourth row can then be found in JA between the 6th and 7th locations (showing that columns 3 and 4 are set).

3.1 Storage Mechanisms

Selection of the storage mechanisms used for the data structures in ENAMeL is important, as this can have a great bearing on the time and space complexity of each operation. The first consideration is the method used to store the positions of the bits set to one.

The most simple method that may be employed is an explicit binary representation, for example `010001010000`. This can provide fast access to specific bits in a vector or matrix, as they can be directly addressed. On the other hand, storing matrices using an explicit representation quickly becomes impractical as the dimensions of the matrix increase.

An alternative method is therefore used in ENAMeL—a compact representation. This representation stores only the positions of the bits set to one, the previous example vector would be stored as {`1, 5, 7`}. In addition to these positions the length of the vector must be stored, as this cannot be calculated. In order to store matrices, a binary version of the Yale format is used [21]. This requires two arrays—IA denotes the beginning of each row of the matrix as an index into the other array, JA, which holds the column indices of bits set to one. This is shown in the example in Fig. 1.

When storing sparse vectors and matrices, this compact representation can greatly reduce the memory required allowing far larger matrices to be used. On the other hand, bits cannot be directly accessed which increases the computational complexity of some operations from constant time (direct access) to logarithmic (binary search). Importantly, the time complexity of a recall operation is not adversely affected by the use of a compact representation—a row can be located and extracted in constant time using the IA index array.

4 ENAMeL Syntax

This is a list of available operations within ENAMeL, providing an explanation of the syntax and output types. Although some explanations refer to operands as vectors or matrices, in ENAMeL they can be used interchangeably—with the operand's types being converted internally as necessary.

Some of the operations are not strictly required, as they could be performed using a series of other basic operations. They are included as extensions to the language, however, as they are commonly required in CMM-based architectures and can be performed more efficiently internally.

Generator $\mathbf{l}_0 \ldots \mathbf{l}_n$ Create a "generator" that uses the algorithm presented by Baum et al. [19] to generate vectors with weight n and length $\sum_{i=0}^{n} l_i$. The vectors are divided into n sections of length l_i, each with a single bit set to one.

A=Vector w l Generate a vector A with weight w and length l, using the appropriate generator.

A=Pattern l $\mathbf{x}_0 \ldots \mathbf{x}_n$ Create a vector A, with length l, weight n, with bits $x_0 \ldots x_n$ set to one. This allows abitrary external vector generation algorithms to be used.

A=Vector B Create a vector A by converting matrix B into a single-row vector in a row-major order.

A=Matrix r c Create a matrix A with rows r and columns c.

A=Matrix r c B Create a matrix A with rows r and columns c by resizing B.

A=B:C Bind vector B to vector C, to form their outer product, storing the result as matrix A.

A=B.C Recall vector B from matrix C, storing the result as non-binary vector A.

A=B.C Recall matrix B from matrix C, storing the result as non-binary matrix A. In this case each column of B will be the same length as an input vector for matrix C. As such, each column will be recalled in turn—the output non-binary vector will be placed in the same position in A as the input vector was taken from B.

A=B,C,n Recall matrix B from matrix C as above, applying Willshaw's threshold function with a value of n to each recalled vector. Each of the recalled binary vectors are then summed together, resulting in a single non-binary vector output.

A=BvC Superimpose B and C, storing the result in A.

A=B+C Sum B and C, storing the result in A. The result of a sum operation is non-binary.

A=B|n Apply Willshaw's threshold function to B with a value of n, storing the result in A.

A=B/n Apply the L-max threshold function to B with a desired output weight of n, storing the result in A.

A=B\n Apply the L-wta threshold function to B with an output weight of n, storing the result in A. The lengths of individual sections of the vector are taken from an appropriate generator.

A=B#n Extract the n^{th} column from matrix B, storing the result as vector A (column indexing is zero-based).

A=n#B Extract the n^{th} row from matrix B, storing the result as vector A (row indexing is zero-based).

a=Weight B Calculate the weight of B, storing the result as integer a.

A=Transpose B Transpose the matrix B, storing the result as matrix A.

A=Append B C ... Append the vectors B, C, etc, storing the result as vector A.

A=B Duplicate B, storing the result as A.

Print A Print the contents of A to the standard output stream, using a compact representation.

4.1 Combining Commands

The binary operations can be combined for improved performance or reduced quantity of code, using parentheses. For example to associate two vectors B and C in a pre-existing CMM M, the explicit instructions would be **A=B:C** followed by **M=MvA**. To avoid the overhead of storing the temporary CMM A and then merging this with M, the combined command **M=Mv(B:C)** can instead be used.

Another common example is the recall of a vector B from a CMM M, followed immediately by a threshold operation. Instead of **A=B.M** followed by **A=A|n**, the combination **A=(B.M)|n** can be used. Not all combinations of commands will improve the efficiency of execution, however they can reduce the quantity of code required and help to improve the "self-documenting" nature of code written using this DSL.

5 Associative Rule Chaining Architecture

Rule chaining and tree searches are used in many fields, for instance artificial intelligence—searching a set of rules to determine if there is a path from the starting state to the goal state. We have presented the Associative Rule Chaining Architecture (ARCA) as an architecture which performs rule chaining or tree search using correlation matrix memories [1]. This architecture uses superposition of search states in order to allow a depth-first search to be performed with a time complexity of $O(d)$, an exponential improvement over traditional methods such as depth-first search with have a time complexity of $O(b^d)$—where d is the depth of the tree and b is the branching factor, the number of children of a node.

5.1 Rule Chaining

Rule chaining encompasses both forward and backward chaining [22], both of which try to determine if there is a route between the starting state and the goal state. The main difference between them is that forward chaining attempts a search beginning at the starting state and working towards the goal, whereas backward chaining begins the search at the goal and performs the search in reverse—the choice of which to use is generally dependent on the specific application. In this work we have concentrated on forward chaining, however there is no reason why ARCA could not be applied to backward chaining instead.

In forward chaining, the search is initialised by creating an initial set of conditions that are known to be true. All of the rules are then searched in turn, to find one for which the antecedents match these conditions. The consequents of that rule can be added to the current state, before checking if the goal state has been reached. If it has not, then the search will continue by searching the rules to find one which matches the new state. If all of the branches of the rule tree have been searched without finding the goal state, then it can be deduced that there is no path from the initial state to the goal state, and hence the search will complete as a failure.

5.2 Associative Rule Chaining

When using Willshaw's method of thresholding, binary CMMs have the unusual property of continuing to operate correctly when input vectors are superimposed [23]. The resultant output vector then contains the superposition of the expected output vectors. As an illustration of this, consider the example set of tokens given in Table 1, and rules in Fig. 2.

Having trained these rules into a CMM, a recall of the token b (or `000100100000`) results in an output of `201102110000`. After applying Willshaw's threshold with a value of 2 (the weight of an input vector), we recover the vector `100001000000`—the expected token d. Similarly, recalling e (or `000010000100`) results in an output of `020000200000`, or `010000100000` after a threshold—again this is the expected token i.

Table 1 An example set of tokens, with a fixed-weight binary vector allocated to each token. These binary vectors were allocated using Baum's algorithm [19] with a length of 12 and weight of 2 (with partition lengths 5 and 7).

Token	Binary vector	Token	Binary vector	Token	Binary vector
a	000100001000	e	000010000100	i	010000100000
b	000100100000	f	100000000010	j	000010010000
c	001000010000	g	010000000001	k	100000001000
d	100001000000	h	001001000000	l	100000100000

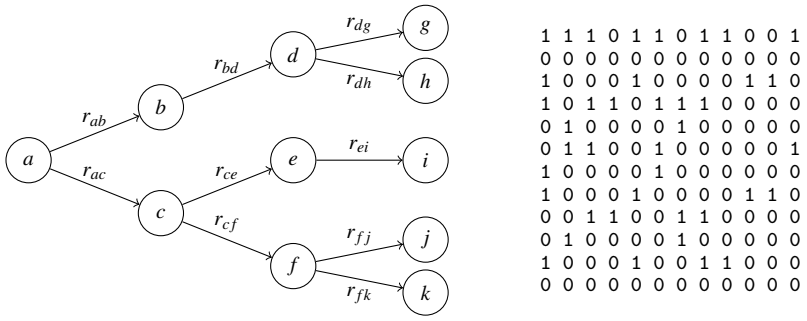

Fig. 2 An example set of rules represented as a tree (on the left), where the nodes represent tokens and the edges represented associations between them. The associations are labelled for ease of future reference. In this tree of rules the branching factor is 2, meaning that each node has at least one and at most two children. On the right is a CMM that has been trained with these rules, using the binary representations given in Table 1 and the method given in equation 1.

The superposition of these input vectors, $b \vee e$, is 000110100100. The result of recalling this from the CMM is 221102310000; it can be clearly seen that after applying the threshold, the vector 110001100000 contains the superposition of both of the expected outputs, $d \vee i$.

Due to overlap between the vectors, however, it can also be seen that the vector l is present in the superimposed output. When using a distributed encoding, it is often not possible to determine which individual vectors are actually present in a superimposed vector, and which simply appear to be present due to overlap with other vectors.

5.3 Tensor Product representation

To describe the contents of a CMM or tensor product, without resorting to the use of binary matrices, we use a pictorial representation. A tensor product is formed by calculating the outer product between two binary vectors, and can be represented as $a : b$. As such it stores the associations between those binary vectors in the same way that a CMM does—in fact a CMM is essentially the superposition of a number of tensor products. Using the example tokens given in Table 1, this tensor product is shown at the top-left of Fig. 3. Below this matrix is our higher-level representation of a tensor product—each displayed column is labelled at the top with the input token it contains, and at the bottom with the output token with which the input was tensored.

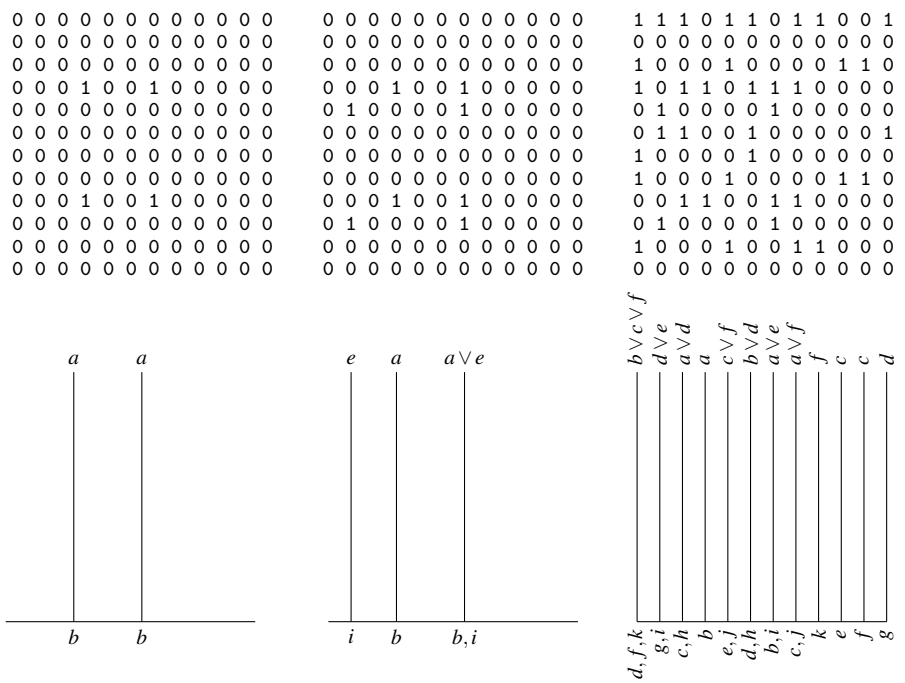

Fig. 3 Three tensor products, or CMMs, given first in matrix form (above) and then in a pictorial representation (below). On the left is the tensor product formed between a and b; formally this tensor product is represented as $a : b$. In the centre is the superposition of this tensor product with that formed between e and i; formally this shows $(a : b) \vee (e : i)$. On the right is the CMM formed by training all of the rules found in Fig 2 using the method given in equation 1. In all of these diagrams, the presence of a particular input vector within a column is indicated by the label at the top of that column. The output vector with which this input was tensored is labelled at the bottom of each column.

In the centre of the figure is the result of superimposing this first tensor product with $e : i$. As there is overlap between the output vectors b and i, the seventh column contains both vectors a and e superimposed.

Finally, on the right of the figure, is our original matrix from Fig 2. Although the matrix contains quite a number of trained vector pairs, it is still possible to quickly determine which columns contain a particular vector.

Columns which are not displayed in a tensor product are assumed to be filled entirely with zeros; it is possible that they contain extraneous noise, however this is irrelevant for the purposes of the diagram.

5.4 Architecture

In ARCA we overcome the limitation of distinguishing between superimposed vectors by assigning each rule a unique "rule vector", taken from a vector space that is distinct to the vectors used for each token.

We separate the antecedents and consequents of each rule into two CMMs, and connect them with this rule vector. Continuing with our example set of rules, we assign each rule a binary vector as shown in Table 2.

Table 2 An example set of rules, with a fixed-weight binary vector allocated to each rule. These binary vectors were allocated using Baum's algorithm [19] with a length of 9 and weight of 2 (with partition lengths 4 and 5).

Rule	Binary vector	Rule	Binary vector	Rule	Binary vector
r_{ab}	100010000	r_{cf}	100000001	r_{fj}	100000010
r_{ac}	010001000	r_{dg}	010010000	r_{fk}	010000001
r_{bd}	001000100	r_{dh}	001001000		
r_{ce}	000100010	r_{ei}	000100100	r_0	001010000

In the antecedent CMM, we associate the antecedents of each rule with its new rule vector—for example $a : r_{ab}$. The rule should then fire if these antecedents are present during a later recall.

The consequent CMM requires a slightly more complex method to train. Firstly, we create a tensor product between the rule vector and the consequents of a rule—for example $b : r_{ab}$. We then "flatten" this tensor product, in a row-major order, resulting in a vector with a length equal to $n_r * n_t$ where n_r and n_t are the respective lengths of a rule and token. The rule vector is now associated with this tensor product and stored in the consequent CMM—essentially storing $r_{ab} : (b : r_{ab})$. This means that recalling a particular rule from this consequent CMM will produce a tensor product containing the output tokens bound to the rule that caused them to fire.

5.5 Recall

Fig. 4 shows two iterations of a recall process performed on our continuing example. In this example, we shall attempt to determine if there is a route from token a to token e.

We must firstly create an input state, containing the tokens with which we are starting our search. A state is simply a tensor product containing tokens bound to rules, and so we bind our input token a to a dummy state r_0 to form TP_{in1}. Each input token will exist in this tensor product a number of times equal to the weight of a rule vector, in our example this is twice.

The first stage of recall is to determine which rules are matched by the tokens contained in our current state. Each column of TP_{in1} must be recalled in turn from the antecedent CMM, resulting in a series of vectors that contain the rule vectors representing any rules which have fired. These vectors are then used as columns to form a new tensor product, TP_{r1}. As can be seen in the figure, each token vector in TP_{in1} has been replaced by a vector containing the rules which fired due to that token, and hence each rule vector will exist twice in the intermediary tensor product (a number of times equal to the weight of a rule vector).

The final stage of an iteration is to find the consequents of the rules which have been matched. To do this, we must recall each column of TP_{r1} in turn from the consequent CMM. In this case, the result of each recall is a "flattened" tensor product containing rules bound to output tokens—we can simply reform each vector to recover a series of tensor products, TP_{out1}. As every rule vector existed twice in TP_{in1}, we know that every output token will be found bound to a particular rule within two of the output tensor products (an equal number to the weight of a rule vector).

Each of the output tensor products has the same dimensions as an entire state, and so we can sum them together to form a non-binary matrix. We then exploit the knowledge that

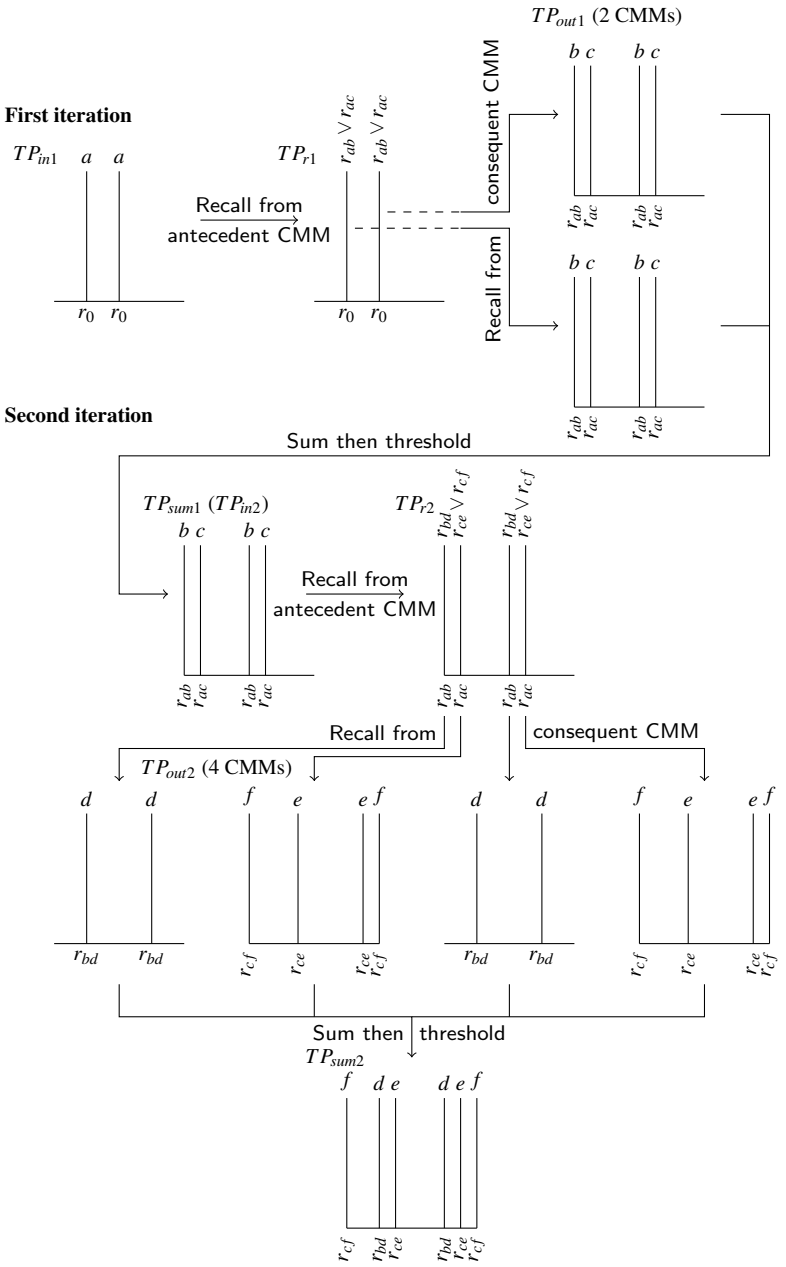

Fig. 4 A visualisation of two iterations of the rule chaining process within ARCA. The process is initialised by creating a tensor product, TP_{in1}, binding the input tokens (in this case *a*) to a rule vector (r_0). Each column of this tensor product is then recalled in turn from the antecedent CMM, to form a new tensor product, TP_{r1}, containing the rules which have fired. Each column of TP_{r1} can then be recalled in turn from the consequent CMM, resulting in a number of output tensor products (TP_{out1}—one tensor product for every non-zero column of TP_{r1}). These output tensor products can be summed, to form a non-binary CMM, before a threshold is applied using a value equal to the weight of a rule vector to form TP_{sum1}. The second iteration continues in the same fashion, using TP_{sum1} as the new input tensor product.

every output token will have been repeated twice and apply this as a threshold in order to reduce the number of erroneously recalled bits, or "ghosts".

Having obtained a single tensor product, this first iteration is complete and we must check whether the search has completed. Firstly we can check whether any rules have been matched, and hence whether the search should continue. If TP_{sum1} consists only of zeros then we know it contains no tokens, and hence the search completed in failure.

If, on the other hand, TP_{sum1} is not empty then we must check whether our goal state has been reached. To achieve this, we consider TP_{sum1} as a CMM. We can then superimpose the goal tokens, and recall this from TP_{sum1} using a threshold equal to the weight of these superimposed tokens. If the result of this recall contains a rule vector, then this rule was bound to the goal tokens and we can conclude that the goal state has been reached. In the absence of any such rule, the system simply iterates.

In our worked example, it is clear that recalling the vector e from TP_{sum1} will not result in an output rule. The second iteration is thus started, using TP_{sum1} as our new input state—TP_{in2}. The operation of the ARCA system continues in exactly the same fashion with each iteration, first recalling each column of the input state from the antecedent CMM to form TP_{r2}, before recalling each column of this intermediary state to result in a series of output tensor products—TP_{out2}. In this iteration, there are four non-zero columns in our input and intermediate tensor products, and so we recover four output tensor products.

To complete the example, we sum and then threshold the TP_{out2} tensor products, and are left with a single output state—TP_{sum2}. In this case, recalling the vector e from our output state will result in a vector containing the rule r_{ce}. Thus we know that our goal state has been reached, and which rule was matched—as it is possible that a token may appear more than once in a particular search tree.

6 Comparison with MATLAB

Our previous results demonstrated that ARCA is able to search multiple branches of a tree simultaneously, while maintaining separation between each of the branches [1]. As the branches are searched simultaneously, the time complexity of this search is reduced from $O(b^d)$ to $O(d)$, where b is the branching factor and d is the depth of the tree. Due to the simple implementation used, however, these results are very limited with regard to high memory usage and experimental execution time—although the number of steps in the search is reduced, each step takes longer than in a classical search.

One of the most important aims of ENAMeL is to ease development and prototyping of applications using CMMs. Due to the domain knowledge employed when designing ENAMeL, however, it is also intended that both the execution time and the memory requirements of these applications be reduced.

6.1 Analysis of a single CMM

When using a non-sparse matrix storage the memory required to store a single matrix is equal to the product of the lengths of the input and output vectors, as shown in equation 3.

$$E = l_{\text{in}} l_{\text{out}} \tag{3}$$

When using the binary Yale format [21] for sparse matrix storage, on the other hand, the memory requirement becomes dependent on the number of bits set to one within the matrix.

In turn, this depends on the number and weight of vectors associated in the CMM. We can formulate an equation for the worst-case memory requirement, by making the assumption that none of the vectors overlap, however this would be too conservative to be of any use. Instead, this overlap is estimated using an equation from [17]. Equation 4 estimates S_t, the number of bits that are set in a matrix after t pairs of vectors have been associated where l and w are the respective lengths and weights of vectors, and subscripts distinguish between inputs and outputs.

$$S_t = l_{\text{in}} l_{\text{out}} \left[1 - \left(1 - \frac{w_{\text{in}} w_{\text{out}}}{l_{\text{in}} l_{\text{out}}} \right)^t \right] \tag{4}$$

Using this estimation, the memory required for a CMM can be calculated as shown in equation 5.

$$\begin{aligned} E_{\text{sparse}} &= \text{size}(\text{IA}) + \text{size}(\text{JA}) \\ &= (l_{\text{in}} + 1) * \text{size}(\text{int}_{\text{IA}}) + l_{\text{in}} l_{\text{out}} \left[1 - \left(1 - \frac{w_{\text{in}} w_{\text{out}}}{l_{\text{in}} l_{\text{out}}} \right)^t \right] * \text{size}(\text{int}_{\text{JA}}) \end{aligned} \tag{5}$$

The number of bits required to store an integer depends on the highest value that may be required to be stored. The IA array stores indices into the JA array, and so is required to be able to store integers with a value equal to the number of bits set in the matrix. If the matrix is entirely full, then this value is equal to the product of the lengths of the input and output vectors, shown in equation 6. The JA array, on the other hand, stores only column positions, and so is required to be able to store integers with a value equal to the number of columns—or the length of the output vector—as in equation 7.

$$\text{size}(\text{int}_{\text{IA}}) = \lceil \log_2(l_{\text{in}} l_{\text{out}}) \rceil \tag{6}$$

$$\text{size}(\text{int}_{\text{JA}}) = \lceil \log_2(l_{\text{out}}) \rceil \tag{7}$$

Although this is the minimum number of bits required to store each integer, for general use it is not actually practical to store arrays in this fashion, due to the high overhead required to store or retrieve these integers. As such, a more useful limit is obtained by rounding the number of bits up to the nearest power of 2, resulting in equations 8 and 9.

$$\text{size}(\text{int}_{\text{IA}}) = 2^{\lceil \log_2(\log_2(l_{\text{in}} l_{\text{out}})) \rceil} \tag{8}$$

$$\text{size}(\text{int}_{\text{JA}}) = 2^{\lceil \log_2(\log_2(l_{\text{out}})) \rceil} \tag{9}$$

Substituting equations 8 and 9 into equation 5 gives us the ability to reasonably calculate the expected storage requirements for a CMM, using a practical number of bits to store each integer, as shown in equation 10.

$$E_{\text{sparse}} = (l_{\text{in}} + 1) 2^{\lceil \log_2(\log_2(l_{\text{in}} l_{\text{out}})) \rceil} + l_{\text{in}} l_{\text{out}} \left[1 - \left(1 - \frac{w_{\text{in}} w_{\text{out}}}{l_{\text{in}} l_{\text{out}}} \right)^t \right] 2^{\lceil \log_2(\log_2(l_{\text{out}})) \rceil} \tag{10}$$

6.2 Analysis of ARCA

To calculate the memory requirements of ARCA, we must simply sum the memory requirements of each of its matrices. The first CMM uses token vectors as inputs, and rule vectors as outputs. The second CMM uses rule vectors as inputs, but the outer product of token and

rule vectors as outputs. Integrating this information gives us two new equations, 11 and 12, where a subscript t refers to a property of a token vector, and a subscript r to a rule vector.

$$E = l_t l_r + l_r(l_t l_r) \tag{11}$$

$$\begin{aligned} E_{\text{sparse}} = {} & (l_t+1)2^{\lceil \log_2(\log_2(l_t l_r))\rceil} + l_t l_r \left[1 - \left(1 - \frac{w_t w_r}{l_t l_r}\right)^t\right] 2^{\lceil \log_2(\log_2(l_r))\rceil} + \\ & (l_r+1)2^{\lceil \log_2(\log_2(l_r(l_t l_r)))\rceil} + l_r(l_t l_r)\left[1 - \left(1 - \frac{w_r(w_t w_r)}{l_r(l_t l_r)}\right)^t\right] 2^{\lceil \log_2(\log_2(l_t l_r))\rceil} \end{aligned} \tag{12}$$

Analysing the relationship between equations 11 and 12 is tricky, due to the large number of variables involved. In our previous work, we limited the experimental scope by using the same length for all vectors, and using $\lfloor \log_2(l) \rfloor$ for the weights of vectors. This allows our equations to be simplified, reducing the variables to only the vector length and the number of vector pairs associated—as shown in equations 13 and 14.

$$E = l^2 + l^3 \tag{13}$$

$$\begin{aligned} E_{\text{sparse}} = {} & (l+1)2^{\lceil \log_2(\log_2(l^2))\rceil} + l^2 \left[1 - \left(1 - \frac{\lfloor \log(l) \rfloor^2}{l^2}\right)^t\right] 2^{\lceil \log_2(\log_2(l))\rceil} \\ & + (l+1)2^{\lceil \log_2(\log_2(l^3))\rceil} + l^3 \left[1 - \left(1 - \frac{\lfloor \log(l) \rfloor^3}{l^3}\right)^t\right] 2^{\lceil \log_2(\log_2(l^2))\rceil} \end{aligned} \tag{14}$$

When using sparse vectors, such that $w \ll l$, the memory requirement of matrices stored using the binary Yale format is very much dependent on the number of vector pairs associated within it. Equation 14 gives us an estimate of the memory required for a given vector length and number of rules that stored.

Table 3 shows this equation applied to a range of the number of rules trained, t and the minimum vector length found experimentally to be able to accurately store and retrieve this many rules. We can see that any memory benefit from using the binary Yale format is expected to be variable, dependent on how close the matrices are to saturation. This result is partly due to the choice of vector weights to use—as the vector length increases, so to does the vector weight, which has a large effect on the memory usage of the binary Yale format.

Considering this from another angle, the memory required to store a CMM using the binary Yale format is essentially the product of the input and output vector weights multiplied by the number of vector pairs associated within it, reduced slightly by the overlap between vectors. Consider an implementation of ARCA storing 300 rules, using a vector length of 100 and weight of 6. Using an explicit matrix storage, this requires 123.29KB of memory, compared with an expected 129.61KB when using the binary Yale format.

Supposing that this system is reaching saturation, it may not be possible to guarantee correct operation. Under these circumstances, and for applications where accuracy is an important factor, it would be advisable to increase the size of the CMMs in order to allow for spare capacity and higher reliability. Increasing the vector length only slightly to 105 can have a significant impact on the memory requirements—it becomes 142.66KB when using an explicit storage, an increase of 15.71%. When using the binary Yale format, on the other hand, the expected memory requirement increases to only 130.45KB, an increase of 0.65%.

Table 3 Table showing the expected memory requirements of ARCA calculated using equation 13 (E), and when stored sparsely calculated using equation 14 (E_{sparse}) for a range of t—the number of rules trained. The minimum successful vector length was found experimentally as the shortest vector length that allowed the ARCA system to accurately store and retrieve the given number of rules. The weight of vectors was selected as $\lfloor \log_2(\text{length}) \rfloor$.

t	Minimum successful vector length	E (KB)	E_{sparse} (KB)
5	32	4.13	1.46
10	32	4.13	2.75
15	32	4.13	4.00
20	32	4.13	5.23
25	32	4.13	6.42
50	40	8.01	12.65
75	48	13.78	19.11
100	60	26.81	25.85
125	72	46.20	54.26
150	72	46.20	64.34
175	88	84.13	76.57
200	88	84.13	86.87

6.3 Experiments

The analysis so far performed has been conservative, and so in order to determine any improvement in memory requirements that may actually result from using ENAMeL we have run the same experiments as we described in our previous work [1]. For this work, however, we used ENAMeL instead of MATLAB and recorded the memory required when storing the CMMs using the sparse binary Yale format.

In each experimental run, a tree of rules was generated with a given depth d and a maximum branching factor b. This maximum branching factor means that the number of children for each node in the tree is uniformly randomly sampled from the range [1, b]. These rules were learned by the system, and rule chaining was performed.

The experiments have been performed over the same range of values for d, b, and vector length, l, as previously for the purposes of a direct comparison. In this set of results, however, we record E_{sparse} rather than simply calculating E.

The experimental methodology is exactly as in our previous work [1]. Briefly, for each value of d, b, and l, we:

1. Generate a rule tree with depth d and maximum branching factor b.
2. Train ARCA with the generated rules, with codes being generated by Baum's algorithm [19]. For each iteration, the starting Baum code is determined randomly.
3. Take the root of the rule tree as the starting token, and select a token in the bottom layer of the tree as the goal token.
4. Perform recall in ARCA with the given starting and goal tokens.
5. Note whether the recall was successful.
6. Repeat the previous steps 100 times.

For each run with a given d, b, and l, we then obtain two results—firstly whether the recall was successful, and secondly the memory requirement of this particular ARCA system. The success rate for recall is thus calculated in the same fashion as before, and the memory requirement for a given d, b, and l is then calculated as the mean over all 100 runs.

The results are shown graphically for various branching factors in Figs. 5 to 9. These are scatter plots that show the memory required to store a rule tree of a given depth, in re-

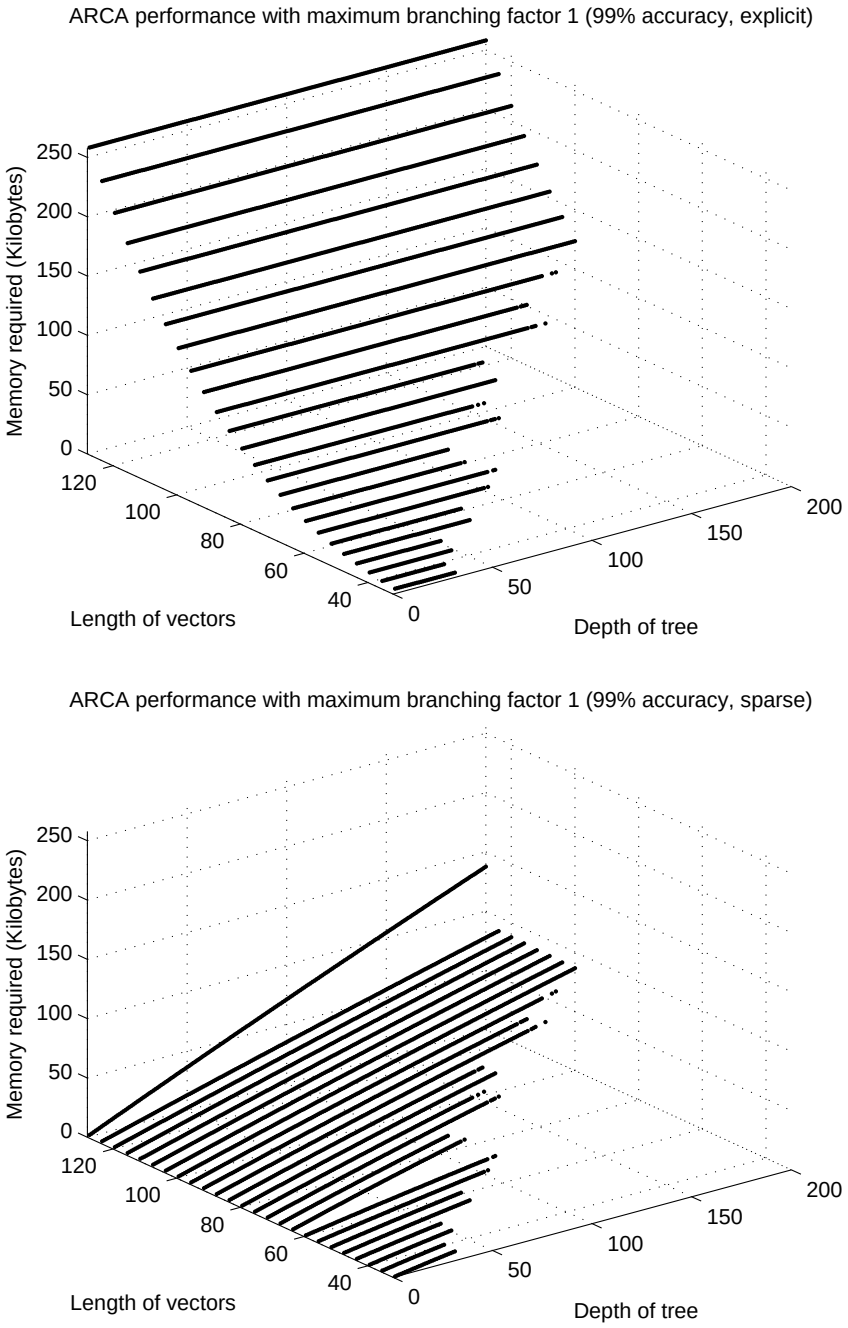

Fig. 5 Scatter plots showing the recall performance for ARCA where the branching factor is 1, with a target of 99% accuracy. At the top is the memory required when storing CMMs explicitly using MATLAB, and at the bottom is the memory required when storing CMMs using the binary Yale format in ENAMeL. Each point represents a successful recall in at least 99 of the 100 experimental runs.

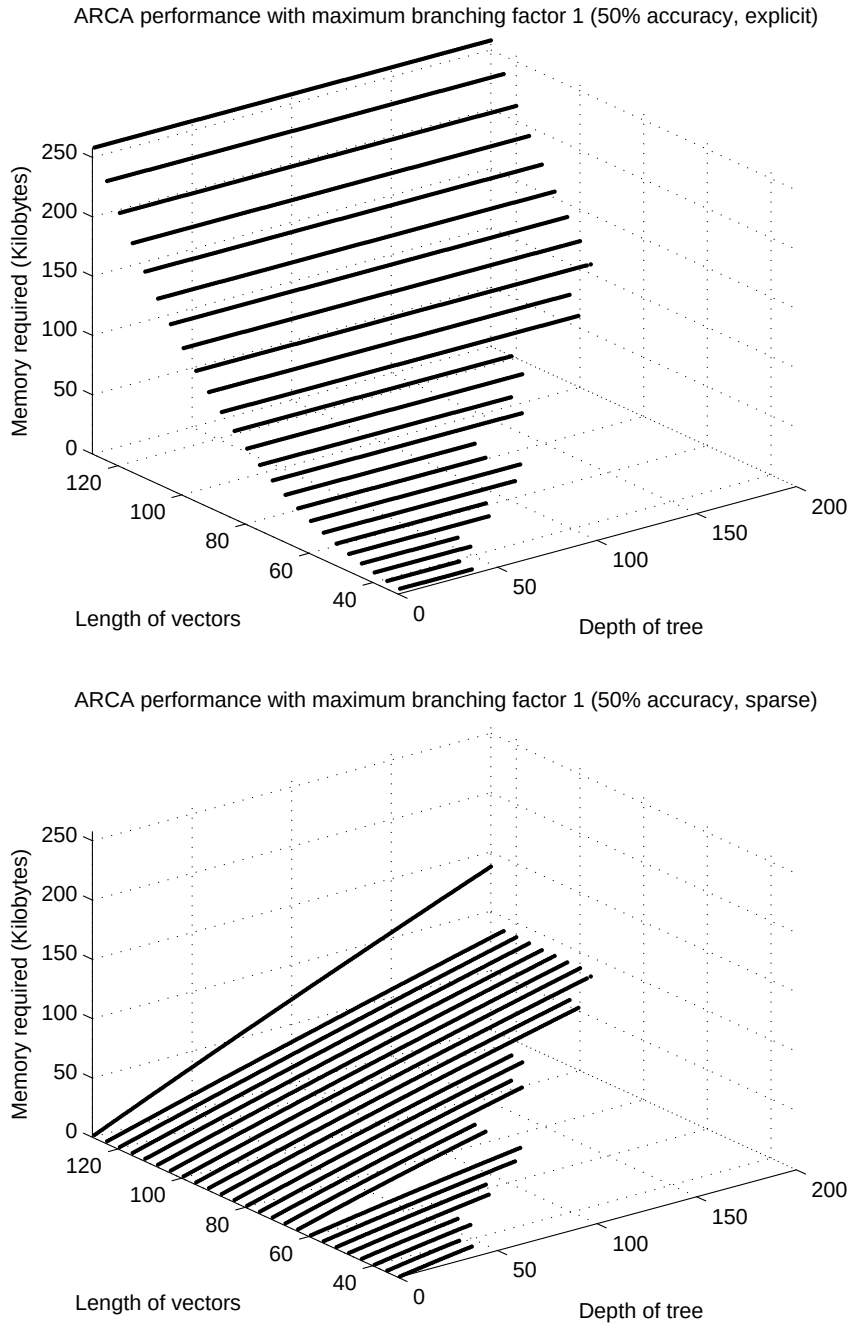

Fig. 6 Scatter plots showing the recall performance for ARCA where the branching factor is 1, with a target of 50% accuracy. At the top is the memory required when storing CMMs explicitly using MATLAB, and at the bottom is the memory required when storing CMMs using the binary Yale format in ENAMeL. Each point represents a successful recall in at least 50 of the 100 experimental runs.

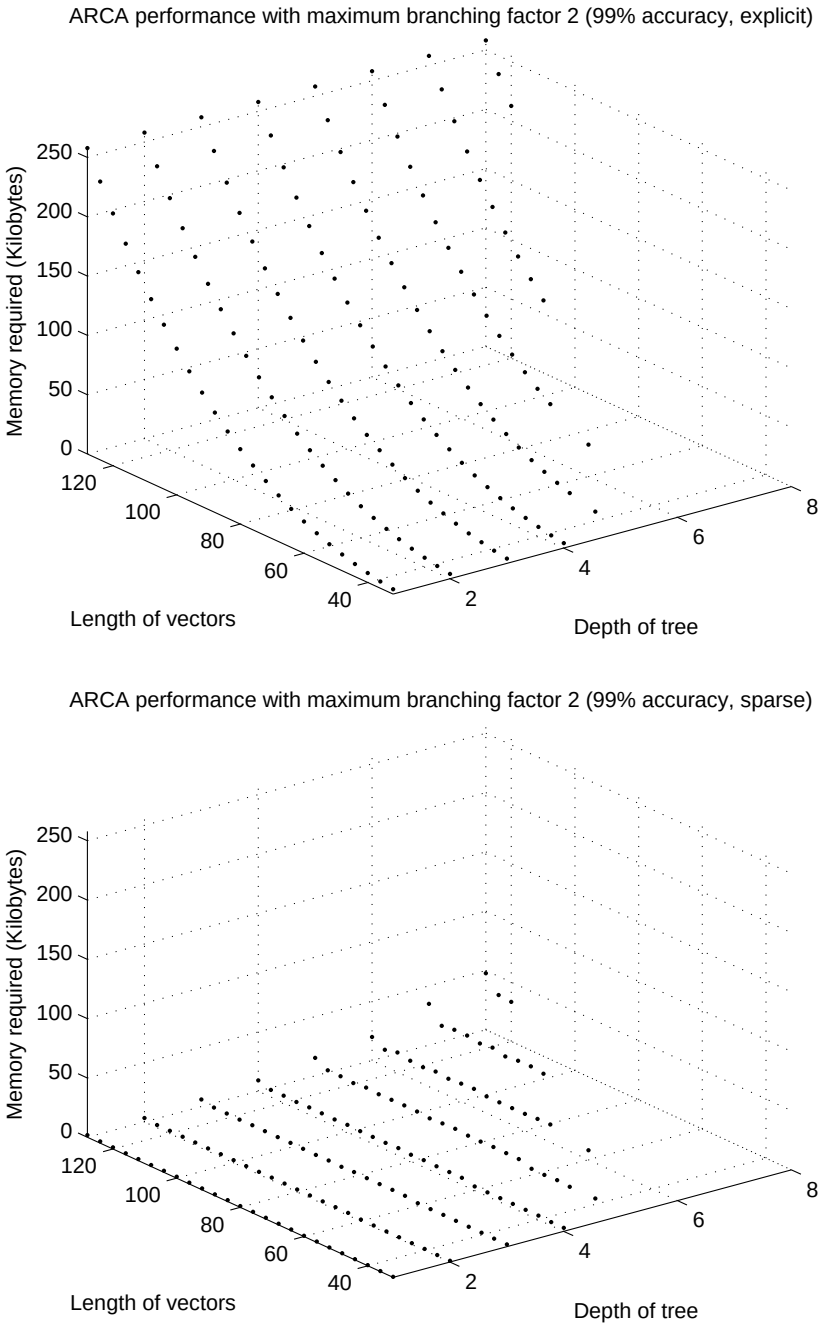

Fig. 7 Scatter plots showing the recall performance for ARCA where the branching factor is 2, with a target of 99% accuracy. At the top is the memory required when storing CMMs explicitly using MATLAB, and at the bottom is the memory required when storing CMMs using the binary Yale format in ENAMeL. Each point represents a successful recall in at least 99 of the 100 experimental runs.

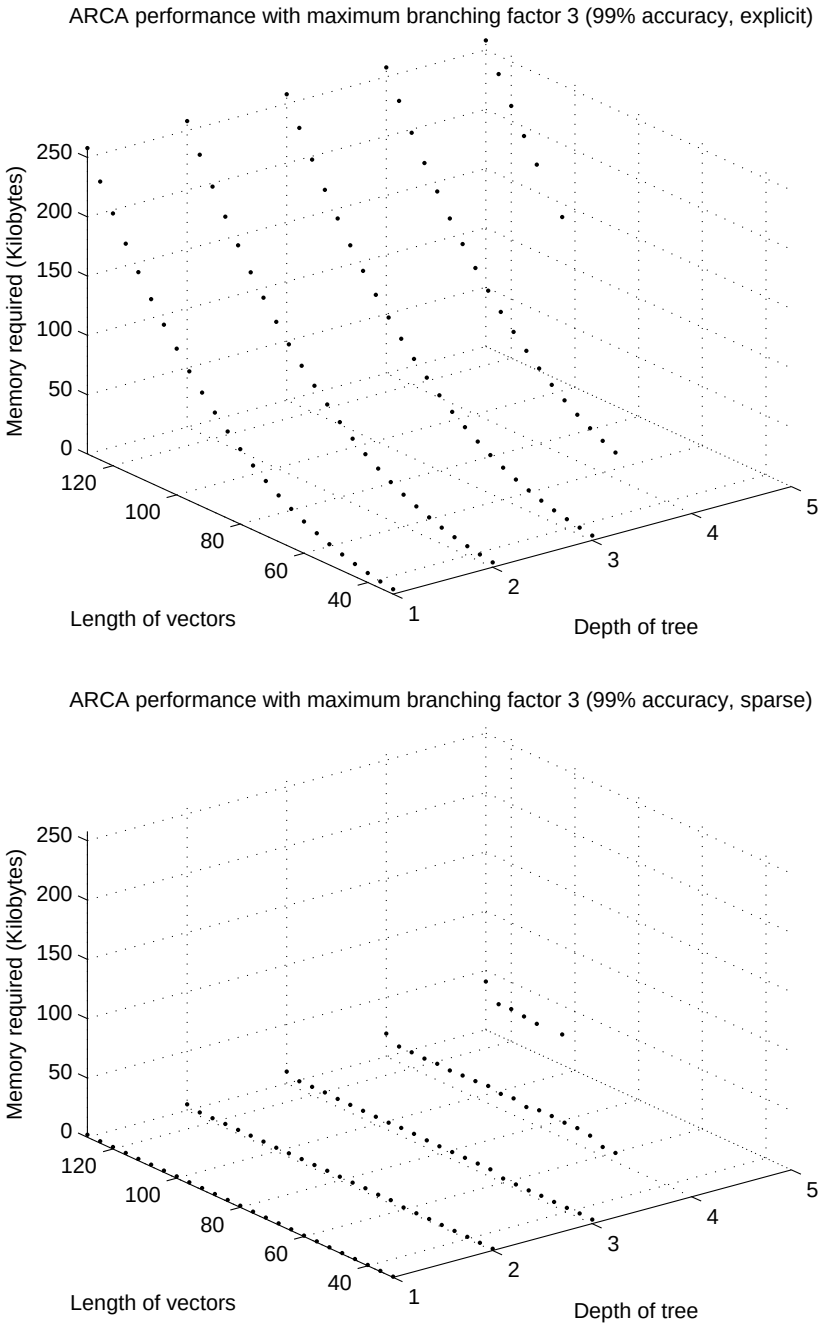

Fig. 8 Scatter plots showing the recall performance for ARCA where the branching factor is 3, with a target of 99% accuracy. At the top is the memory required when storing CMMs explicitly using MATLAB, and at the bottom is the memory required when storing CMMs using the binary Yale format in ENAMeL. Each point represents a successful recall in at least 99 of the 100 experimental runs.

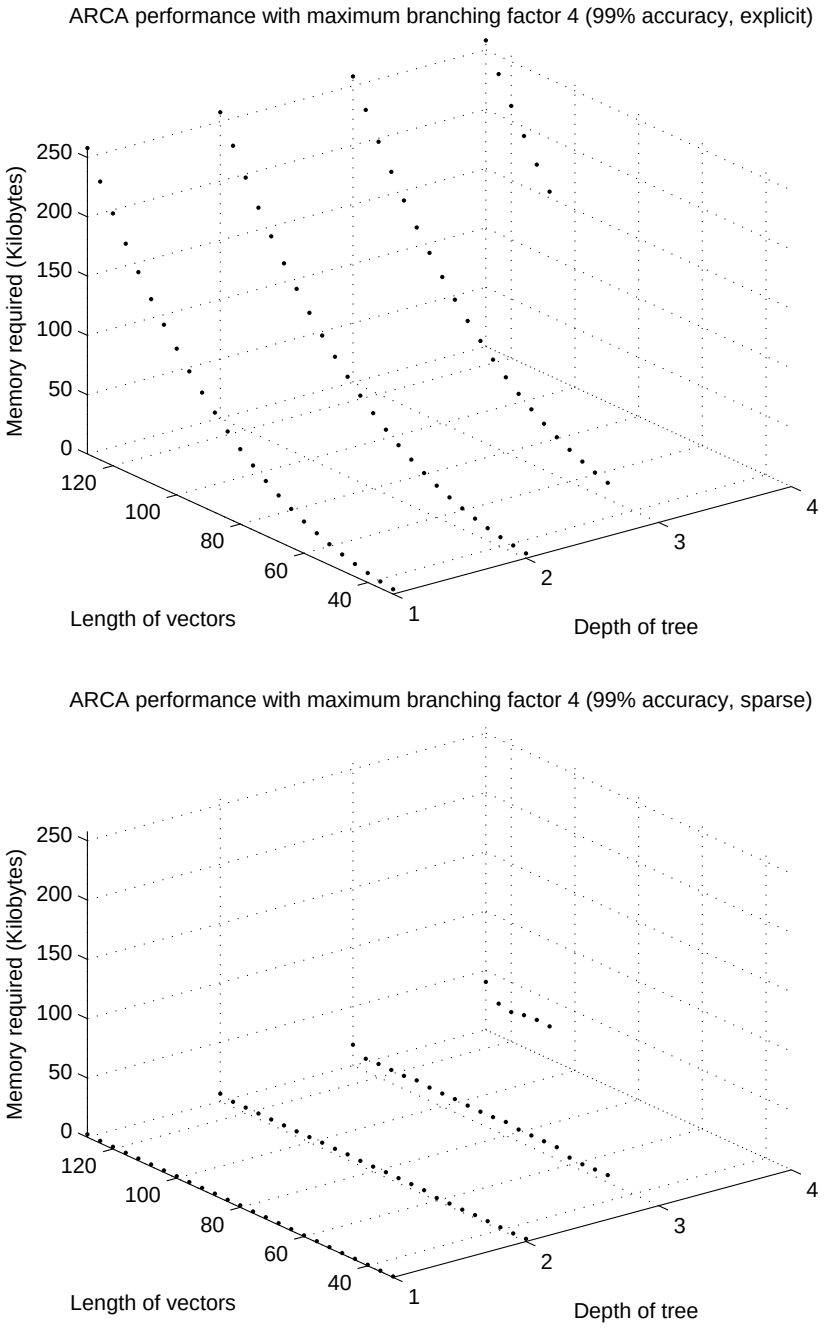

Fig. 9 Scatter plots showing the recall performance for ARCA where the branching factor is 4, with a target of 99% accuracy. At the top is the memory required when storing CMMs explicitly using MATLAB, and at the bottom is the memory required when storing CMMs using the binary Yale format in ENAMeL. Each point represents a successful recall in at least 99 of the 100 experimental runs.

lation to the length used for vectors. Each point represents a recall that is successful to the given accuracy, as a percentage of the total number of runs. Each figure shows the memory requirement when storing matrices explicitly using MATLAB at the top, and the memory requirement when using the binary Yale format in ENAMeL at the bottom. When storing matrices explicitly the memory required is dependent solely on the vector length used, however when storing matrices sparsely the depth of tree is far more important.

In Fig. 5 we show the results when using a branching factor of 1, displaying a point for each set of parameters that resulted in $\leq 1\%$ error. It is clear to see that the memory requirement for any given depth of tree can be significantly reduced when using the binary Yale format for matrix storage. The sudden increases in gradient are caused by the increase in the vector weight as the vector length moves from 60 to 64 and again from 124 to 128. An increased weight causes a greater number of bits to be set for every additional rule trained.

If Fig. 6 it can be seen that increasing the tolerance of errors to show values resulting in $\leq 50\%$ error has the expected effect of increasing the depth of tree that may be successfully stored with a given vector length. As this is the case with all the different branching factors, we only show the results at this target accuracy for a branching factor of 1.

The remaining figures show the results for branching factors of 2, 3, and 4 with an error rate of $\leq 1\%$. Again it is clear that there is a marked improvement to the memory requirements when using the binary Yale format, even more so than with a branching factor of 1.

7 Conclusions and Further Work

This paper has introduced a new domain specific language, ENAMeL, for use with binary CMMs. We have implemented the ARCA using this language, and demonstrated a significant reduction to the memory required to store the matrices. Although increasing the branching factor still leads to an exponential increase in the memory requirements, there is an order of magnitude improvement compared to storing the matrices explicitly. Additionally, due to the benefits of domain specific languages, the implementation is significantly more simple than that required for MATLAB as can be seen in appendix A.

Within the plots of the experiments with a branching factor higher than 1, it is possible to see areas with a distinct gradient, where the almost linear growth is only interrupted by an increase in the vector weight. This warrants further investigation as to the effect of fixing the vector weight, rather than setting it as $\log_2$ of the vector length. Additionally, the results indicate that a greater reduction is achieved at higher values of d, b, and l (depth, branching factor, and vector length), although further experimentation would be required in order to confirm this.

Experimentally we have shown that using the binary Yale format for sparse matrix storage can greatly decrease the memory requirement when compared with storing the CMMs explicitly. Our analysis, however, indicated that this may not be true in all circumstances—depending on the number of vector pairs trained into a CMM with a given vector length and weight. The results in Table 4 show that this is indeed the case. Although comparing the plots shows that in general the memory required using sparse storage is far lower than when using explicit storage, when the system begins to reach saturation point (which is the case for the results shown) the memory required by ENAMeL can surpass that required by MATLAB.

Finally, Willshaw et al. determine theoretically that the number of bits set in a CMM may reach as high as 50% of the number of bits within that CMM while still maintaining

Table 4 Table showing the actual memory requirements of ARCA calculated using MATLAB (E), and when stored sparsely calculated using ENAMeL (E_{sparse}) for a range of t—the number of rules trained. The minimum successful vector length was found experimentally as the shortest vector length that allowed the ARCA system to accurately store and retrieve the given number of rules. The weight of vectors was selected as $\lfloor \log_2(\text{length}) \rfloor$. E_{hybrid} is the memory requirement of ARCA when using a very basic hybrid system—each CMM is stored either sparsely or explicitly, depending on which is smallest.

t	Minimum successful vector length	E (KB)	E_{sparse} (KB)	E_{hybrid} (KB)
5	32	4.13	1.47	1.41
10	32	4.13	2.79	2.62
15	32	4.13	4.07	3.79
20	32	4.13	5.32	4.13
25	32	4.13	6.52	4.13
50	40	8.01	12.47	8.01
75	48	13.78	18.77	13.78
100	60	26.81	25.32	23.82
125	72	46.20	53.33	46.20
150	72	46.20	62.89	46.20
175	88	84.13	74.78	71.23
200	88	84.13	84.48	80.56

accurate recall [15]. With such a highly populated CMM, it would not be possible for the binary Yale format to use less memory than an explicit storage. Therefore as further work it would be advantageous if ENAMeL was able to switch between the use of the binary Yale format and explicit storage, selecting which to use in order to minimise the memory required.

Using a hybrid system will cause the memory requirement to be bounded by the explicit storage requirement, as shown in Table 4. E_{hybrid} is the memory required by ARCA for a basic form of hybrid system, where each of the CMMs is stored either explicitly or sparsely depending on which is smallest. Additionally, a hybrid system which was able to use a combination of both formats—in different sections of the CMM—may be able to improve the memory usage even further.

In ENAMeL the binary data structures are defined as abstract classes, to allow other parts of the system to interact with them without needing to know their internal structure. The concrete implementations then store bits using the sparse format. It is therefore possible to write alternative implementations that store bits using the explicit format. This will require some minor logic to be added in any function that creates or modifies a binary data structure, in order that they can select which implementation to use. For example a vector of length 256 with 31 set bits will require less memory when stored sparsely, whereas a vector of length 256 with 33 set bits will require less memory when stored explicitly.

Acknowledgements Our thanks to Ken Lees for his initial work on developing the syntax of ENAMeL. Financial support from the Engineering and Physical Sciences Research Council is gratefully acknowledged.

References

1. Austin J, Hobson S, Burles N, O'Keefe S (2012) A rule chaining architecture using a correlation matrix memory. Int Conf on Artif Neural Netw 2012, pp 49-56. doi: 10.1007/978-3-642-33269-2_7
2. Van Deursen A, Klint P, Visser J (2000) Domain-specific languages: an annotated bibliography. ACM Sigplan Not, pp 26-36. doi: 10.1145/352029.352035

3. Brewer G (2008) Spiking cellular associative neural networks for pattern recognition. PhD Thesis, University of York
4. Shanker KPS, Turner A, Sherly E, Austin J (2010) Sequential data mining using correlation matrix memory. Int Conf on Netw and Inf Technol, pp 470-472. doi: 10.1109/ICNIT.2010.5508469
5. Spinellis D (2001) Notable design patterns for domain-specific languages. J Systems and Software, pp 91-99. doi: 10.1016/S0164-1212(00)00089-3
6. Mernik M, Heering J, Sloane AM (2005) When and how to develop domain-specific languages. ACM Computing Surveys (CSUR), pp 316-344. doi: 10.1145/1118890.1118892
7. Kosar T, Lopez PEM, Barrientos PA, Mernik M (2008) A preliminary study on various implementation approaches of domain-specific language. Information and Software Technology, pp 390-405. doi: 10.1016/j.infsof.2007.04.002
8. Ladd DA, Ramming JC (1994) Two application languages in software production. USENIX Very High Level Languages Symposium Proceeding 1994, pp 169-178.
9. Van Deursan A, Klint P (1998) Little languages: Little maintenance? J Software Maintenance, pp 75-92. doi: 10.1002/(SICI)1096-908X(199803/04)10:2¡75::AID-SMR168¿3.0.CO;2-5
10. Kieburtz RB, McKinney L, Bell JM, Hook J, Kotov A, Lewis J, Oliva DP, Sheard T, Smith I, Walton L (1996) A software engineering experiment in software component generation. 18th Int Conf on Software Engineering, pp 542-552.
11. Basu A (1997) A language-based approach to protocol construction. PhD Thesis, Cornell University
12. Bruce D (1997) What makes a good domain-specific language? APOSTLE, and its approach to parallel discrete event simulation. ACM SIGPLAN Workshop on Domain-Specific Languages, pp 17-35.
13. Menon V, Keshav P (2000) A case for source-level transformations in MATLAB. ACM SIGPLAN Notices, vol 35, pp 53-65. doi: 10.1145/331960.331972
14. Krueger, CW (1992) Software reuse. ACM Computing Surveys (CSUR), vol 24, pp 131-183. doi: 10.1145/130844.130856
15. Willshaw DJ, Buneman OP, Longuet-Higgins HC (1969) Non-holographic associative memory. Nature 222, pp 960-962. doi: 10.1038/222960a0
16. Ritter H, Martinetz T, Schulten K, Barsky D, Tesch M, Kates R (1992) Neural computation and self-organizing maps: an introduction. Addison Wesley, Redwood City
17. Austin J, Stonham TJ (1987) Distributed associative memory for use in scene analysis. Image and Vis Comput 5, pp 251-260. doi: 10.1016/0262-8856(87)90001-1
18. Hobson S, Austin J (2009) Improved storage capacity in correlation matrix memories storing fixed weight codes. Int Conf on Artif Neural Netw 2009, pp 728-736. doi: 10.1007/978-3-642-04274-4_75
19. Baum EB, Moody J, Wilczek F (1988) Internal representations for associative memory. Biol Cybern 59, pp 217-228. doi: 10.1007/BF00332910
20. Orovas C, Austin J (1997) Cellular associative neural networks for image interpretation. Sixth Int Conf on Image Process and its Appl, pp 665-669. doi: 10.1049/cp:19970978
21. Eisenstat SC, Gursky MC, Schultz MH, Sherman AH (1982) Yale sparse matrix package I: The symmetric codes. Int J for Numer Methods in Eng, pp 1145-1151. doi: 10.1002/nme.1620180804
22. Russell SJ, Norvig P, Canny JF, Malik JM, Edwards DD (1995) Artificial intelligence: a modern approach. Prentice Hall, Englewood Cliffs
23. Austin J (1992) Parallel distributed computation. Int Conf on Artif Neural Netw 1992.

A Code used for MATLAB and ENAMeL

During the comparison between MATLAB and ENAMeL, all of the rule trees used were generated by MATLAB. After generating a random rule tree, and allocating binary vectors to each token, MATLAB then generated two files—one providing MATLAB instructions to train and recall the generated rule tree, and the second providing ENAMeL instructions to do the same. This was to ensure that both MATLAB and ENAMeL were storing exactly the same randomly generated rules allowing for a direct comparison between their memory requirements.

Below is a sample of the code generated for one of the rule trees. MATLAB uses one-based indexing, where ENAMeL is zero-based.

MATLAB code

```
% Create CMMs
M1=zeros(60);
M2=zeros(60,3600);
% Create and train vectors
T1=vector(60,[2,11,21,32,45]);
T2=vector(60,[4,14,22,34,48]);
...
R1=vector(60,[1,15,25,32,47]);
...
M1=train(M1,T1,R1);
...
M2=traintp(M2,R1,T2,R1);
...
% Recall vectors
% Initialise recall
O2=train(zeros(60),T1,R1);
% Iterate through each level of the tree
% Level 1
O1=recall(M1,O2);
O2=reshape(sum(recall(M2,O1),2),60,60)';
O2(O2<5)=0;
O2(O2>0)=1;
% Level 2
...
```

ENAMeL code

```
# Create CMMs
M1=Matrix 60 60
M2=Matrix 60 3600
# Create and train vectors
T1=Pattern 60 1 10 20 31 44
T2=Pattern 60 3 13 21 33 47
...
R1=Pattern 60 0 14 24 31 46
...
M1=M1 v (T1:R1)
...
M2=M2 v (R1:(T2:R1))
...
# Recall vectors
# Initialise recall
O2=T1:R1
# Iterate through each level of the tree
# Level 1
O1=O2.M1|5
O2=O1,M2,5|5
# Level 2
...
```

MATLAB Functions

```
% Recall vector from CMM
function out=recall(mat,in)
  out=zeros(size(mat,2),size(in,2));
  count=1;
  for col=in
    out(:,count)=sum(mat(col==1,:));
    count=count+1;
  end
  out(out<5)=0;
  out(out>0)=1;

% Train vectors into CMM
function mat=train(mat,in,out)
  mat(:,out==1)=mat(:,out==1)|repmat(in,1,sum(out));

% Train vector / tensor product into CMM
function mat=traintp(mat,in,tpin,tpout)
  tp=train(zeros(size(tpin),size(tpout),tpin,tpout)');
  mat=train(mat,in,reshape(tp,numel(tp),1));

% Create vector with the provided length and set bits
function v=vector(l,bits)
  v=zeros(l,1);
  v(bits)=1;
```