A Rule Chaining Architecture Using a Correlation Matrix Memory

James Austin, Stephen Hobson, Nathan Burles, and Simon O'Keefe

Advanced Computer Architectures Group,
Department of Computer Science,
University of York,
York, YO10 5GH, UK
{austin,stephen,nburles,sok}@cs.york.ac.uk
http://www.cs.york.ac.uk

Abstract. This paper describes an architecture based on superimposed distributed representations and distributed associative memories which is capable of performing rule chaining. The use of a distributed representation allows the system to utilise memory efficiently, and the use of superposition reduces the time complexity of a tree search to $O(d)$, where d is the depth of the tree. Our experimental results show that the architecture is capable of rule chaining effectively, but that further investigation is needed to address capacity considerations.

Keywords: rule chaining, correlation matrix memory, associative memory, distributed representation, parallel distributed computation.

1 Introduction

Rule chaining is commonly used in artificial intelligence, for searching a set of rules to determine if there is a path from the starting state to the goal state. This paper presents the Associative Rule Chaining Architecture (ARCA), which performs rule chaining using correlation matrix memories (CMMs)—a type of simple associative neural network [1]. Biologically inspired systems can provide effective alternatives to classical systems. Rule chaining is thought to be performed by the brain, and so an approach that uses neural networks has the potential to efficiently solve this problem.

This work, for the first time, introduces a method where data is stored in a distributed representation throughout the process, which provides an efficient use of memory and greater possibility for fault tolerance than a local representation [2]. The CMMs form a state machine, as in previous work on Parallel Distributed Computation [3], and are able to store multiple states in a single vector using the distributed representation. The approach given here builds on work undertaken by Kustrin and Austin [4], which showed a simple reasoning system operating in a fully distributed manner.

A.E.P. Villa et al. (Eds.): ICANN 2012, Part I, LNCS 7552, pp. 49–56, 2012.
© Springer-Verlag Berlin Heidelberg 2012

1.1 Rule Chaining

Rule chaining may be approached using forward chaining or backward chaining; the choice of which to use is application specific. The approach we describe here uses forward chaining, although there is no reason that the same techniques could not be used to implement backward chaining.

In forward chaining, starting with an initial set of conditions, all of the rules are searched to find one for which the antecedents match the conditions. The consequents of that rule can then be added to the current state, and the state is checked to decide if the goal has been reached. If the goal state is not yet reached, then the system will iterate. Finally, if all of the branches have been searched without finding the goal state, then the search will complete as a failure. A common way to implement such a search would be with depth-first search. This technique has time complexity $O(b^d)$, where b is the branching factor and d is the depth of the tree. Our technique offers the possibility of significantly improving on this complexity.

1.2 Correlation Matrix Memories (CMMs)

The CMM is a simple neural network consisting of a single layer of weights, where the input and output neurons are fully connected. In this work a sub-class of CMMs is used, known as binary CMMs [5], where these weights are binary.

Binary CMMs use simple Hebbian learning [6]. Learning to associate one binary vector with another is thus an efficient operation, and requires only local updates to the CMM. This training is formalised in Equation 1, where M is the resulting CMM (matrix of weights), x is the set of input vectors, y is the set of output vectors, and n is the number of training pairs.

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

A recall operation is performed as shown in Equation 2. It is essentially a matrix multiplication between the transposed input vector and the CMM. The matrix multiplication produces a non-binary output vector, to which a threshold function f must be applied in order to produce the final output vector.

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

There are various thresholding functions that may be applied during recall. The choice of which function to use depends on the application, and on the data representation used. Willshaw thresholding is used in ARCA, where any output bit with a value at least equal to the (fixed) trained input weight is set to one [5] (the weight of a vector is the number of bits within it that are set to one).

2 Associative Rule Chaining

The aim of this system is to perform rule chaining using superimposed representations, and hence to reduce the time complexity of a tree search. The main

Table 1. An example set of rules with a binary rule vector allocated to each, and tokens with their binary representations

Rule	Rule vector
$r_0 : a \to b$	10100
$r_1 : a \to c$	01010
$r_2 : b \to p$	10001
$r_3 : c \to q$	01001

Token	Binary representation
a	1001000
b	0100100
c	0010001
p	0100001
q	0010100

challenge is thus to maintain the separation of each state throughout the search, without needing to separate out the distributed patterns.

As an example, consider a single CMM containing the rules given in Table 1. In a conventional system, the search would evaluate each of these rules in turn. When using a CMM, however, input of token a would match two rules simultaneously and result in both b and c being superimposed in the output. After applying a threshold, the output would be 0110101—the superposition of vectors b and c. However, given this encoding, this is ambiguous as it could instead be the superposition of vectors p and q.

2.1 Architecture

To resolve this difficulty, each rule is assigned a unique "rule vector", existing in a separate vector space to the token vectors. ARCA separates the antecedents and consequents of the rules into two CMMs, using the rule vector to connect them. The basic rule chaining is performed by a simple state machine formed with two CMMs, as shown in Fig. 1.

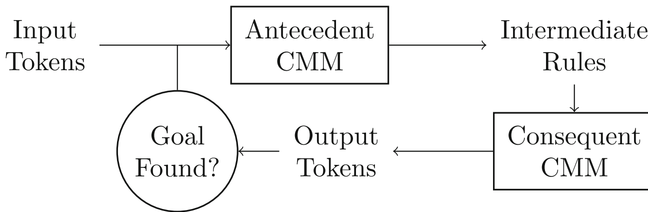

Fig. 1. Block diagram of the Associative Rule Chaining Architecture (ARCA)

The first CMM is trained with the associations between the superimposed antecedents of a rule and the rule vector. This means that the rule should fire if the tokens in the head of each rule are present in an input.

To train the second CMM, a slightly more complex method is used. Firstly, a tensor product (a matrix) is formed between the rule vector and the superimposed consequents of a rule. This tensor product is "flattened" into a vector with a length equal to $n_r * n_t$ where n_r and n_t are the lengths of a rule and token

respectively. The associations between this tensor product and the rule vector are then stored in the CMM. This means that when a rule fires from the antecedent CMM, the consequent CMM will produce a tensor product containing the output tokens bound to the rule that caused them to fire.

2.2 Recall

Fig. 2 shows the recall process performed on part of the continuing example. To begin the recall, an initial state TP_{in} is created by forming the tensor (outer) product of any initial tokens with a rule vector. In the diagram we initialise the search with two tokens b and c, bound to the appropriate rules.

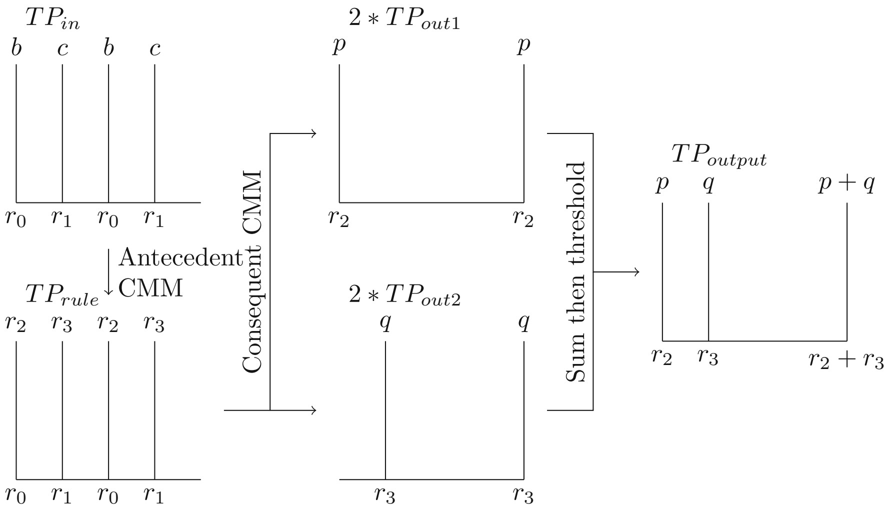

Fig. 2. A visualisation of the recall process within ARCA. The tensor products contain different token vectors bound to rule vectors. Each column is labelled at the top with the tokens contained within the column. The position of each column is defined by the positions of the bits set to one in the rule vector to which the token is bound, and this is labelled at the base of the column. The remainder of the tensor product consists solely of zeros. The vector weight is 2, and hence each column appears twice.

The first stage of recall is now to determine which rules are matched by the tokens in the current state. To do this, each column of TP_{in} is recalled from the antecedent CMM in turn. The result of each column recall is a vector that contains the rule vectors representing any of the matched rules. The recalled vectors form the columns of an intermediary tensor product TP_{rule}.

To continue the recall, each column of TP_{rule} is recalled from the consequent CMM in turn. Remember that the result of a recall from the consequent CMM is effectively a reshaped tensor product containing the consequents of a rule, bound to the rule that fired them. Therefore each recalled column will result

in an entire tensor product of equal dimensions to the original input—TP_{out1} and TP_{out2}. Note that each of these tensor products will be recalled twice, once for each column in TP_{rule}. We wish to reduce these to a single tensor product ready to iterate, and so we sum them to form a non-binary tensor product before applying another threshold.

As can be seen in the diagram, any rule will appear in TP_{rule} a number of times equal to the weight of a rule vector. Thus, the consequents bound to that rule will appear in the same number of TP_{out}s. When these are summed, the weight of a rule vector can therefore be used as a threshold to obtain the final output—a single binary tensor product. Note that in the example we can observe that TP_{output} contains $p : r_2$ and $q : r_3$ (where ":" means "bound to"), precisely the result we would expect.

The final stage of recall, before the system iterates, is to check whether the search is completed. Firstly we can check whether any rules have been matched, and therefore whether the search should continue. This is achieved by checking whether TP_{output} contains any non-zero values. If TP_{output} consists solely of zeros, then no rules have been matched and hence the search is completed without finding a goal state.

If TP_{output} is not empty, then we must check whether a goal state has been reached. This is achieved by treating TP_{output} as a CMM. The superposition of the goal tokens is used as an input to this CMM, and the threshold set to the product of the number of goal tokens and the weight of those tokens. If the resulting binary vector contains a rule vector, then this indicates that this rule vector was bound to the goal token and so we can conclude that the goal state has been reached.

3 Time and Space Complexity

We have demonstrated that ARCA is able to search multiple branches of a tree in parallel, while maintaining separation between them. This means that the time complexity of a search becomes $O(d)$, where d is the depth of the tree. Contrasted with a depth-first approach with a time complexity of $O(b^d)$, where b is the branching factor, this is a notable improvement.

On the other hand there is frequently a trade-off between time complexity and space complexity, and this is no different in the case of ARCA. While a depth-first search has space complexity of $O(bd)$, comparison with ARCA is a complicated issue. Although the space used by the system is constant throughout operation, if insufficient space is initially allocated then the memories will become saturated and recall will begin to fail. The space complexity can thus be defined as the amount of memory required to perform an error free recall. Unfortunately the storage capacity of a CMM cannot be easily predetermined because of the distributed encoding; with further work we are aiming to improve on our estimation, and so will be able to provide a more detailed comparison. Finally, it should also be noted that as the size of a CMM increases, so does the time required to perform operations on it.

4 Experiments

In order to show ARCA working on search trees, and to determine at what point the method fails, a number of experiments have been performed. ARCA has been implemented in MATLAB, and applied to a variety of problems. For each experiment a tree of rules was generated with a given depth d and maximum branching factor b. These rules were then learned by the system, and rule chaining was performed on them.

The experiments have been performed over a range of values for d, b, and the memory required to implement the CMMs in ARCA, E. This allows some limited comparison between the system and a depth-first search.

Search trees were constructed in an iterative manner, beginning with the root token, which was also used as the starting token for the recall process. Further layers of the tree were then added, with the total number of layers being d. In the simple case of $b = 1$, this produces a chain of rules $A \rightarrow B$, $B \rightarrow C$, etc. In the case of $b > 1$, the number of children of a given node was uniformly randomly sampled from the range $[1, b]$. This results in a tree with maximum branching factor b which is more realistic than one in which all nodes have exactly b children.

E is determined by the token and rule vector lengths n_t and n_r, with the required memory being $n_r n_t + n_r^2 n_t$. To simplify the experiment, the weight of all vectors was set to $\log_2 n$ (rounding down), where n is the vector length. This value gives a sparse representation, and should provide good performance in the CMMs [7]. For each value of d, b, and E, the following experiment was performed:

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 [2]. For each iteration, the starting Baum code was 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.

This gives a success rate for recall in ARCA, for a given combination of d, b, and E. A recall was defined as successful if and only if the goal token was found at the correct depth (i.e. after d iterations of the system). In unsuccessful recall, the system does not "fail" in a traditional manner. Often the system will arrive at the desired goal erroneously, due to saturation of the CMMs causing extra patterns not trained to be recalled ("ghosts"). In these cases, the recall was determined to have been unsuccessful.

The graphs in Fig. 3 are contour plots showing the recall error rates for the ARCA architecture for a given depth of search tree and memory requirement. They clearly demonstrate that the ARCA system is capable of performing rule chaining in a fully distributed, superimposed manner. However they also show

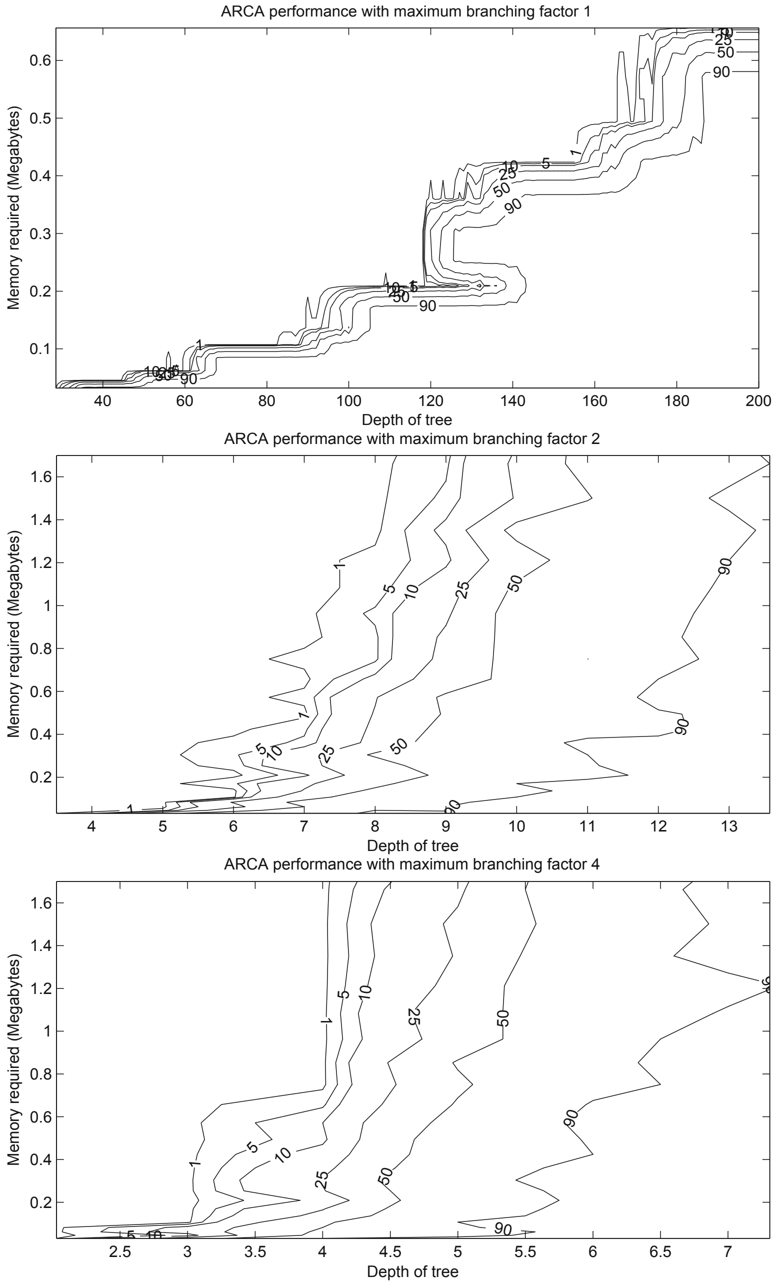

Fig. 3. Contour plots showing the recall error performance for ARCA where the branching factor is 1 (top), 2 (middle), and 4 (bottom). Contours summarise the recall error at various depths and memory requirements, effectively showing the percentage of recalls which are in error for a given size of CMM and depth of tree. Discontinuity in the contours is caused by a change in the vector weight (selected as $\log_2 n$).

a decrease in performance as the branching factor increases. This is to be expected given the explosion in the number of rules that occurs, and is similar to that experienced by traditional methods such as depth-first search—leading to their time complexity of $O(b^d)$. In this case, however, the explosion leads to exponentially increasing memory requirements rather than running time.

5 Conclusions and Further Work

This paper has introduced a novel architecture capable of performing forward chaining by examining multiple branches of a tree simultaneously. Experimentation has shown that the architecture is capable of rule chaining effectively, but the capacity considerations warrant further investigation.

The choice of the length of vectors used to represent tokens and rules in ARCA are very important, as these define the memory requirement. While both of these values must be large enough to allow all of the tokens and rules in the system to be represented, further work is required to understand the effect that varying the lengths separately has on recall performance. In addition, the weight chosen may not be optimal and so this needs further investigation.

Finally, in this examination of ARCA we have only considered the case where each rule has a single symbol as the antecedent; arity one rules. This allowed a clear and simple examination of the architecture, but the system will be extended to handle multiple arity rules in further work. We are considering looking into applying the technique to games, initially simple ones such as tic-tac-toe.

References

1. Kohonen, T.: Correlation Matrix Memories. IEEE Transactions on Computers, 353–359 (1972)
2. Baum, E.B., Moody, J., Wilczek, F.: Internal Representations for Associative Memory. Biol. Cybernetics 59, 217–228 (1988)
3. Austin, J.: Parallel Distributed Computation in Vision. In: IEE Colloquium on Neural Networks for Image Processing Applications, pp. 3/1–3/3 (1992)
4. Kustrin, D., Austin, J.: Connectionist Propositional Logic A Simple Correlation Matrix Memory Based Reasoning System. In: Wermter, S., Austin, J., Willshaw, D.J. (eds.) Emergent Neural Computational Architectures Based on Neuroscience. LNCS (LNAI), vol. 2036, pp. 534–546. Springer, Heidelberg (2001)
5. Willshaw, D.J., Buneman, O.P., Longuet-Higgins, H.C.: Non-holographic Associative Memory. Nature 222, 960–962 (1969)
6. Ritter, H., Martinetz, T., Schulten, K., Barsky, D., Tesch, M., Kates, R.: Neural Computation and Self-Organizing Maps: An Introduction. Addison Wesley, Redwood City (1992)
7. Palm, G.: On the Storage Capacity of Associative Memories. In: Neural Assemblies, an Alternative Approach to Artificial Intelligence, pp. 192–199. Springer, New York (1982)