BoltzCONS: Dynamic Symbol Structures in a Connectionist Network

David S. Touretzky
School of Computer Science, Carnegie Mellon University, Pittsburgh, PA 15213, USA

ABSTRACT

BoltzCONS is a connectionist model that dynamically creates and manipulates composite symbol structures. These structures are implemented using a functional analog of linked lists, but BoltzCONS employs distributed representations and associative retrieval in place of a conventional memory organization. Associative retrieval leads to some interesting properties, e.g., the model can instantaneously access any uniquely-named internal node of a tree. But the point of the work is not to reimplement linked lists in some peculiar new way; it is to show how neural networks can exhibit compositionality and distal access (the ability to reference a complex structure via an abbreviated tag), two properties that distinguish symbol processing from lower-level cognitive functions such as pattern recognition. Unlike certain other neural net models, BoltzCONS represents objects as a collection of superimposed activity patterns rather than as a set of weights. It can therefore create new structured objects dynamically, without reliance on iterative training procedures, without rehearsal of previously-learned patterns, and without resorting to grandmother cells.

1. Introduction

BoltzCONS[1] is a neural network that dynamically creates and manipulates composite symbol structures, such as stacks and trees. In LISP, these structures are represented as linked lists. In BoltzCONS, we investigate what a parallel, distributed version of linked lists might look like. The goal is not to arrive at some peculiar new version of LISP, or to suggest that any representation as impoverished as cons cells might actually exist in the brain. Rather, it is to address the issues of compositionality, cited by Fodor and Pylyshyn [5], and what Newell [14] calls distal access, that help to distinguish symbol processing from lower-level cognitive functions like pattern recognition.

"Compositionality" is the recursive combining of symbol structures into larger, more complex structures. It is an essential feature of language. "Distal access" is the ability to reference a structure in some remote, abbreviated way, such as via a pointer or symbolic tag. Without this ability, concepts would have

[1] The name is a play on Boltzmann machines [9] and CONS, the first MIT LISP Machine [6].

Artificial Intelligence **46** (1990) 5–46

to be written out in full detail everywhere they were referenced. Limited resources, and the circularity of semantic representations, preclude this. Compositionality therefore requires distal access. These issues are fundamental ones which connectionist systems must deal with if they are to address the full range of human cognitive phenomena, rather than being limited to pattern recognition and associative memory [25].

It is an open question whether symbolic data structures such as frames, parse trees, and semantic nets have any cognitive validity. Presently, though, it is difficult to imagine a comprehensive cognitive theory without such structures. BoltzCONS is an attempt at reconciling the functional properties of these data structures with the implementational constraints of PDP models [22]. Not all connectionists concede the necessity of such a reconciliation. For example, Rumelhart and McClelland's verb learning model [18] maps input strings to output strings in a way that captures both the rules of English past tense formation and the many classes of exceptions, yet they explicitly deny that the model has symbolic rules or a lexicon. Pinker and Prince [15] refer to this as "eliminative connectionism," because it eliminates the symbolic level as a valid level of description. Symbolic theories, according to the eliminativists, are no more than crude approximations to what really takes place in the brain. They are not a truthful high-level description of the neurological facts in the way that the source listing of a Pascal program can be a truthful description of the machine language version. This anti-symbolic view is controversial, but it has yet to be effectively refuted.

McClelland, Rumelhart, and Hinton [12], in arguing against the validity of symbol structures, suggest that what we perceive introspectively to be symbolic processes are mere epiphenomena of an underlying subsymbolic system. Smolensky [20] characterizes the subsymbolic level as a continuous dynamical system. The evolution of its states through time may be well-approximated by a discrete symbolic theory,[2] but complete accuracy would only be achievable by descriptions phrased as numerical differential equations. He goes on to suggest that what we consciously experience as discrete throughts may be snapshots of the dynamical state vector taken when a number of processing units have remained stable for a few tens of milliseconds.

If the extreme eliminativist view is correct, the long struggle to symbolically axiomatize such things as deep and surface-level linguistic structure, episodic memory, goals, beliefs, and so forth, can never succeed. At the other extreme, one could adopt the "implementationalist" stance [15] that discrete symbolic representations are perfectly adequate, and that connectionist networks are just another implementation technology, not a new theoretical approach. I don't see how either of these views can be correct. My goal is to explore how the properties of a connectionist implementation influence our understanding

[2]This admission separates him from the radical eliminativist camp.

of what symbol processing is about.

Despite their support elsewhere for an eliminativist view, Hinton, Rumelhart and McClelland [9, p.78] warn that "it would be wrong to view distributed representations as an *alternative* to representational schemes like semantic networks or production systems...." Instead they suggest that parallel, distributed processing models can implement these schemes in ways that have important emergent properties. These properties would distinguish connectionist networks from other implementations of symbol processing theories. In Pinker and Prince's taxonomy [15], this position is called "revisionist-symbol-processing connectionism." BoltzCONS is an example of this approach.

Many of the differences between the way BoltzCONS and a von Neumann machine process data are due to the use of parallel associative retrieval. Associative retrieval is not unique to connectionist models. One can always duplicate its functionality (though not its efficiency) on a conventional computer using sequential search. And one can sometimes even obtain the same efficiency, by using hash tables. There are, however, certain areas where the decision to use a connectionist architecture, as opposed to some other parallel model not constrained to resemble neurons, uniquely influences the choice of representations and the efficiency of primitive operations. Elucidating those influences is the primary contribution of this paper.

What distinguishes BoltzCONS from many earlier connectionist models is its ability to construct and modify composite symbol structures *dynamically*, by representing them as activity patterns rather than as weights. BoltzCONS is therefore not limited to retrieving one of a set of pre-existing patterns. It can create new ones "on the fly," without extensive training, without rehearsal of previously learned patterns to prevent their decay, and without resorting to grandmother cells.

The control of BoltzCONS is external to the model. While it would not be difficult to build a finite state machine from simulated neurons to issue the necessary control signals for copying activity patterns from one module to another, initiating an associative retrieval, and so on, this would add little of interest. The real issues the model addresses are issues of representation.

Internally, BoltzCONS uses coarse-coded, distributed representations for linked lists that are quite unlike von Neumann machine data structures. It includes a functional equivalent of pointers, but no notion of addresses. Its associative retrieval capabilities support primitives that are not available in conventional computer implementations of linked lists. One modest example is instantaneous access to the internal nodes of a tree, given a node label. But *connectionist* associative retrieval suggests much more powerful operations, such as rapidly accessing parts of a symbol structure based on closest match rather than exact match. This puts BoltzCONS-style models in the realm of truly revisionist symbol processing, pointing the way toward new computational theories that exploit the special strengths of PDP architectures.

2. Direct and Indirect Representations

Both stacks and trees are special cases of directed graphs. LISP offers a direct representation for a highly restricted class of graphs as analogous cons cell structures. By *direct representation* I mean that each element of the graph corresponds to either an atom or a cons, and the basic graph operations of finding the left or right child of a node, and constructing a new nonterminal node given its two children, correspond to the LISP primitives car, cdr, and cons. Only directed graphs whose nodes and links are unlabeled, and whose nonterminal nodes all have out-degree 2, can be represented this way. Stacks and binary trees fall in this category, but general tree structures do not.

Different versions of BoltzCONS also offer natural, direct representations for various kinds of graphs. The version primarily discussed in this article is called BoltzCONS-3. It can directly represent the same set of structures as LISP, although its repertoire of primitive operations is larger. Section 5.3 discusses another version of the model, called BoltzCONS-5, that has a richer direct representation.

In order to manipulate graphs for which no direct representation is available, programmers employ *indirect representations*. For example, one often needs to represent general (not strictly binary) trees. A common technique is to represent each nonterminal node by a linked list of its children, marking the cell that points to the last child by placing something special in its cdr. If nodes are labeled, then the first cell of the linked list holds the node label; the remaining elements point to the children. Interpreted LISP programs are represented as tree structures in precisely this way.

Following this convention, the tree of Fig. 1 would be represented in LISP by the list shown in Fig. 2. Internally, this list is a cons structure as shown in Fig. 3. The slashes in the cdrs of some cells indicate the termination of a cons cell chain. In LISP, the termination marker is the distinguished symbol **nil.**

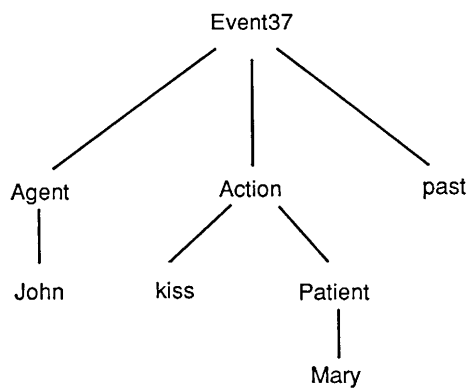

Fig. 1. A tree with labeled nonterminals.

```
(Event37
  (Agent
    John)
  (Action
    kiss
    (Patient
      Mary))
  past)
```

Fig. 2. Linked list representation of the tree of Fig. 1.

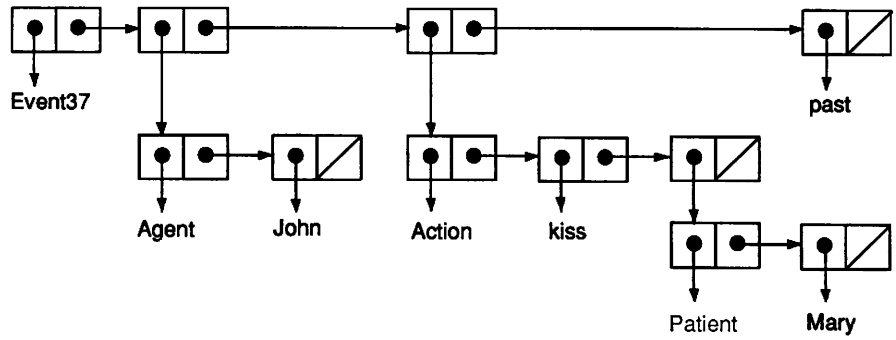

Fig. 3. Cons cell representation of the tree of Fig. 1.

The major disadvantage of this indirect representation for trees is that it limits the ways one can access the nodes. For example, on a von Neumann machine one cannot access the rightmost child of a node in constant time. Another problem is that given a pointer to an arbitrary node in a tree, one can find its descendants but not its parents or siblings, because von Neumann machines cannot follow pointers backward. LISP programmers are of course free to create more complex indirect representations that overcome these limitations, but doing so would increase the cost of representing and modifying the tree, and block the use of LISP's many built-in tree manipulation primitives.

3. Representing Linked Lists on an Associative Retrieval Machine

This and the two following sections give a high-level overview of the BoltzCONS architecture as an abstract associative retrieval machine. Section 6 then presents the connectionist implementation of BoltzCONS, including details of the distributed representation and the wiring patterns of the various modules.

TAG	CAR	CDR
p	Event37	q
q	r	t
r	Agent	s
s	John	s
t	u	z
u	Action	v
v	kiss	w
w	x	w
x	Patient	y
y	Mary	y
z	past	z

Fig. 4. The linked list structure of Fig. 3 encoded as tuples.

3.1. Encoding cells as tuples

We can represent one cell of a linked list as a three-tuple of symbols of form **(tag, car, cdr)**. Tags serve as the targets of what would be called pointers in conventional computers. But tags are symbols, not addresses. In particular, they are not integer indices into a vector of sequential memory locations, as on a von Neumann machine. The memory of our abstract associative retrieval machine has nothing corresponding to discrete sequential addresses.

The symbols in the car and cdr fields of a tuple refer either to the tags of other cells, or to atoms (terminal nodes; objects without composite structure). No two cells may have the same tag. Figure 4 shows one way the linked list structure of Fig. 3 could be encoded as tuples. Other ways are possible, since the assignment of tags is arbitrary. This encoding strategy does not use **nil** to mark the end of chains, due to a property of the distributed memory representation, to be described later.

3.2. An architecture for associative retrieval

A general outline of the BoltzCONS architecture is shown in Fig. 5. Tuple Memory contains the set of tuples that encode the graph structures BoltzCONS creates. Tuple Buffer holds only a single tuple, known as the "current tuple." Sometimes it is empty. The individual components of the current tuple, if there is one, are also represented in the three symbol spaces labeled TAG, CAR, and CDR. These spaces can be clamped (meaning their state is frozen) and used to drive associative retrievals from Tuple Memory via the buffer. For example, if Tuple Memory contains the set of tuples in Fig. 4, then clamping the symbol ***p*** into TAG space and performing an associative retrieval would cause (***p***, **Event37,** ***q***) to appear in the Tuple Buffer. Simultaneously, **Event37** and ***q*** would appear in the CAR and CDR spaces, respectively. (The actual simulation uses just the symbols *A* through *Y*. I use symbols like John and Event37 here to distinguish atoms from tags, and to suggest that atoms might

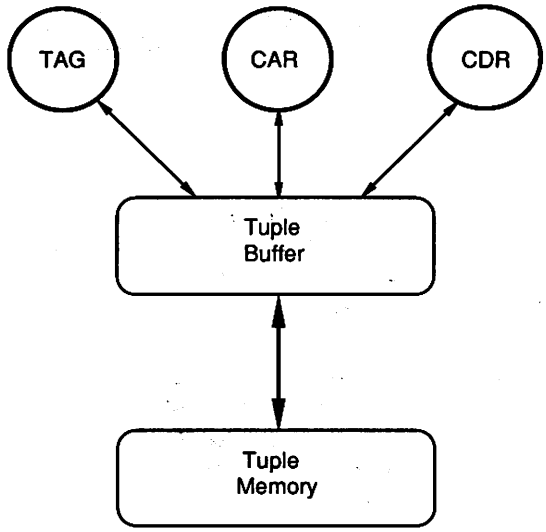

Fig. 5. The BoltzCONS architecture.

have semantic content, even though BoltzCONS itself does not rely on this content.)

3.3. Pointer traversal

Pointer traversal by associative retrieval is straightforward. We assume there is a current tuple in the Tuple Buffer, with its components represented in TAG, CAR, and CDR spaces. LISP's car function is implemented by copying the symbol currently in CAR space into TAG space, clamping TAG space, clearing the CAR and CDR spaces and the Tuple Buffer, and performing an associative retrieval. This fetches the triple with the specified TAG value into the Tuple Buffer; simultaneously, its second and third components are fetched into CAR and CDR space, respectively.

At this point it will be convenient to introduce a notation for sequences of these operations. The retrieval just described consists of two steps in this notation:

```
GetCAR =
  {$CAR→TAG;
   $Retrieve.by.TAG}
```

The first step of the GetCAR sequence is to issue a control signal causing the symbol in CAR space to be copied into TAG space. This destroys the previous contents of TAG space. The second step issues a control signal that initiates an associative retrieval, with TAG space providing the retrieval cue. The units in

TAG space are clamped so they cannot change during the retrieval. The BoltzCONS implementation of LISP's cdr function is similar:

```
GetCDR =
  {$CDR→TAG;
   $Retrieve.by.TAG}
```

To use pointer traversal to go from the **Event37** node of Fig. 1, represented by the tuple (*p*, **Event37**, *q*), to the Action node, represented by (*u*, **Action**, *v*), a sequence of two cdrs followed by a car is required. A suitable composition of the GetCAR and GetCDR procedures is easily constructed, since each procedure leaves its result in the Tuple Buffer and associated symbol spaces, where it may serve as the argument to the next procedure.

With associative retrieval one is not limited to following pointers in the forward direction. We can go from the **Agent** node of Fig. 1, represented by the tuple (*r*, **Agent**, *s*), to its parent node, by performing an un-car operation. The first step is to copy the symbol *r* from TAG space to CAR space. An associative retrieval with CAR space clamped then yields the tuple (*q*, *r*, *t*). We have followed a pointer backward from the cons cell *r* to the cons cell *q*. Next an un-cdr is performed. The symbol *q* is copied from TAG space and clamped into CDR space, and after a second associative retrieval, (*p*, **Event37**, *q*) appears in the Tuple Buffer.

```
un-CAR =
  {$TAG→CAR;
   $Retrieve.by.CAR}

un-CDR =
  {$TAG→CDR;
   $Retrieve.by.CDR}
```

These little procedures are in some ways analogous to Ullman's notion of visual routines [30]. They are short, simple routines, with direct hardware implementations, that form the primitives from which more complex processing operations are built.

3.4. Detecting atoms and list termination

There are several ways one might distinguish terminals from nonterminals (or atoms from cells). One way is to *a priori* divide the set of symbols into those that may be used as tags for composite objects and those that may not. The latter class may then be used to refer to atoms. A minor drawback to implementing this approach is that the model must somehow be able to tell which class each symbol is in.

A second approach is to note that a symbol associated with an atom is not the tag of any composite object, so it will never appear as the first component

of any tuple. An associative retrieval with that symbol clamped into TAG space will fail, i.e., whatever it retrieves will not match the specified tag. Thus the model can determine whether a symbol refers to a composite object by attempting one associative retrieval. The drawback to this method is that extra associative retrivals waste time, and performing one will destroy the state of the unclamped symbol spaces. If the retrieval fails, their state may need to be restored before the next step of the computation can proceed.

A third approach is to represent atoms as tuples with a special marker in all but the first field, e.g., the symbol John could be represented by the tuple (**John**, *, *). The model can simply check for the presence of the * marker after a tuple is retrieved to determine whether it represents an atom. But this method would not work well in BoltzCONS due to the model's coarse-coded, distributed memory representation. In a phenomenon called local blurring, when a distinguished symbol appears in the same position in many tuples, it reduces the accuracy of the distributed memory. This is one of the ways in which the connectionist implementation influences the design of the model.

A fourth approach represents atoms as tuples whose car and cdr fields contain the atom's own tag. This eliminates the local blurring problem, since each atom will have unique car and cdr values. In this scheme, John would be coded as (**John, John, John**). It is easy to detect when a tuple represents an atom: the model simply compares the tag, car, and cdr fields to see if they are identical.

In applying BoltzCONS to various problems, both the first and fourth methods for distinguishing atoms have been used.

A related concern is the method of marking termination of linked lists. Although **nil** could be used as a terminator, following the LISP convention, it would have to appear many times in highly-branched structures, which raises the probability of local blurring interfering with the accuracy of retrieval. An alternative is to mark the last cell in a chain by having its cdr point to itself, as in Fig. 4. The model can easily detect a cell marked this way because its tag and cdr components are equal. If this convention is used for list termination, then one of the other three conventions must be used to distinguish atoms from composite objects. In the example in Fig. 3, we assume that the symbol space has been divided *a priori* into symbols that denote atoms and symbols that may be used as tags for cells.

3.5. Creating new structure

We create new list cells by adding tuples to Tuple Memory. The first step in adding a tuple is to clamp values into the TAG, CAR, and CDR spaces. These are then assembled into a new tuple in the Tuple Buffer. Any previous value in the Tuple Buffer is discarded. Finally, the pattern in the Tuple Buffer is added to the contents of Tuple Memory.

In the procedure MakeCell below, certain operations can proceed in parallel because they involve independent modules or transmission paths. These operations appear on the same line, separated by ampersands, to highlight the potential parallelism. The parameters x and y represent inputs from external symbol spaces that are not part of the BoltzCONS model, but are connected to it in the context of some larger information processing architecture. An example of such an architecture is given in [23].

```
MakeCell(x, y) =
  {$x→CAR & $y→CDR & $NewTag→TAG;
   $Assemble.tuple.in.buffer;
   $Store.tuple}
```

The problems of avoiding collisions when choosing tags for new tuples, and reclaiming tags no longer in use (garbage collection), will be addressed in Section 7.

3.6. Modifying structures

LISP destructively modifies cells by storing new pointers into the car or cdr half with the rplaca and rplacd operations, respectively. In BoltzCONS the equivalent effect can be achieved by deleting the tuple and storing another with the modified components. For example, to change the agent of Fig. 1 from John to Bill, the tuple (s, **John,** s) would be called into the Tuple Buffer and then deleted from Tuple Memory. Then the triple (s, **Bill,** s) would be assembled in the buffer and stored in the memory. Due to the distributed representations BoltzCONS uses for its Tuple Memory, the order of these operations is important: the delete operation should take place before the store. The procedures below assume that the tuple to be modified is the one currently represented in the Tuple Buffer, with its components represented in the TAG, CAR, and CDR spaces.

```
ReplaceCAR(x) =
  {$Delete.tuple.from.memory & $x→CAR;
   $Assemble.tuple.in.buffer;
   $Store.tuple}

ReplaceCDR(x) =
  {$Delete.tuple.from.memory & $x→CDR;
   $Assemble.tuple.in.buffer;
   $Store.tuple}
```

The delete operation in the above procedures can be done in parallel with the transfer of a new symbol into CAR or CDR space, because deletion is performed by the Tuple Buffer and only affects the state of Tuple Memory.

4. Associative Stacks

This and the following section present associative versions of two familiar recursive data structures: stacks and trees.[3] In both cases the use of associative retrieval leads to slightly different algorithms with different performance characteristics than LISP on von Neumann machines.

Stacks may be represented as linked lists. The top of the stack resides in the Tuple Buffer, and also in the TAG, CAR, and CDR spaces. The stack is popped by taking its cdr, i.e., deleting the tuple currently in the Tuple Buffer from Tuple Memory, copying the symbol in CDR space to TAG space, and doing an associative retrieval with TAG space clamped. The new top of the stack then appears in the Tuple Buffer. An empty stack may be denoted by a tuple with a special "top of stack" marker as its car component. This tuple will always be the last one in the chain. The procedures below do not check for empty stack or stack full conditions.

```
StackPush(x) =
  {$TAG→CDR & $x→CAR;
   $NewTag→TAG;
   $Assemble.tuple.in.buffer;
   $Store.tuple}

StackPop =
  {$Delete.tuple.from.memory & $CDR→TAG;
   $Retrieve.by.TAG}
```

Notice that a **NewTag** operation is used by the stack push procedure to generate a new tag for the cons that is about to become the top of the stack. One way to avoid the problem of generating new tags dynamically when building stacks is to construct a static linked list that is as long as the maximum desired stack depth. The cell that is the top of the stack is maintained in the Tuple Buffer, as before. (The initial contents of the Tuple Buffer will be the *last* cell of the chain.) To push a new object onto this fixed stack, we use associative retrieval to find the cell that points to the current one, make it current, and store the new object into its car:

```
FixedStackPush(x) =
  {un-CDR;
   ReplaceCAR(x)}

FixedStackPop =
  {GetCDR}
```

[3] A recursive data structure is one whose instances are of the same type as their components. Trees are recursive because their branches are trees. Stacks are recursive because their tails are stacks.

Associative retrieval permits another interesting stack operation: associative stack pop. This operation pops the stack back to the point where a specified element is the top, in constant time. If the element appears on the stack more than once, an instance can be picked at random. The items above the one being sought are not deleted, so one can find the top of the stack again by repeatedly un-popping it until the un-pop operation fails. In order to detect when a retrieval has failed, we introduce a "verify" operation to confirm that the retrieval has found a tuple whose specified component exactly matches the cue supplied.

```
AssocPop(x) =
  {$x→CAR;
   $Retrieve.by.CAR}

UnPop( ) =
  {un-CDR}

FindStackTop =
  {loop
    unPop;
    $Verify.CDR.retrieval;
    if $Retrieval.failed      — unpopped once too many;
      then exit-loop          — TAG and CAR hold garbage
    endloop;
    GetCDR}                   — CDR still valid; undo the failed UnPop
```

This technique for returning to the top of the stack won't work for fixed stacks because cells aren't deleted when the stack is popped; the current top of stack must be marked somehow before an associative stack pop in order to permit it to be found again. Any number of marking conventions may be employed. For example, cells above the current top of the stack may have their car set equal to their tag, so the sequential unpop operation can tell when it has gone too far.

5. Associative Trees

In describing operations on tree structures, one must be careful to distinguish between trees with direct representations (i.e., binary trees whose internal nodes are unlabeled), and more general sorts of trees. We will consider each type in turn.

5.1. Traversing binary trees

Associative retrieval allows one to nondestructively traverse binary trees of

unbounded depth without a control stack. LISP cannot do this.[4] The procedures shown below are tail-recursive, and hence can be implemented by a finite state machine. The top level procedure, Traverse, traverses a binary tree and outputs the symbols at the terminal nodes in left-to-right order. (A neural network can "output" a symbol by transmitting its activity pattern to some external module to which the network is connected.)

The Traverse procedure assumes that the root of the tree is initially the current tuple, and that atoms are represented by conses whose cdrs point to themselves. The details of how the root is remembered so the algorithm knows when to exit are omitted; a simple marking convention or an auxiliary register may be used.

```
Traverse =
  {$Remember.root;
   DownCAR}

DownCAR =
  {if $TAG.neq.CDR then  —at a nonterminal node
     GetCAR;
     DownCAR
   else                  —at a terminal node
     $Transmit.CAR;
     Backup
   endif}

DownCDR =
  {if $TAG.neq.CDR then  —at a nonterminal node
     GetCDR;
     DownCAR
   else                  —at a terminal node
     $Transmit.CAR;
     Backup
   endif}

Backup =
  {un-CAR;
   $Verify.CAR.retrieval;
   if $Retrieval.succeeded then  —was parent's car
     DownCDR
   else                          —wasn't parent's car, so
```

[4] We assume one-way pointers, as in normal LISP lists. If destructive operations are allowed, then LISP can traverse binary trees without a control stack; this is the basis of certain sophisticated garbage collection algorithms.

```
    $CAR→CDR;                  — must be parent's cdr
    $Retrieve.by.CDR;
    if $TAG.eql.root then      — quit if parent is root
      exit
    else                       — continue backing up
      Backup
    endif
  endif}
```

One subtlety in the above algorithm is the method of backing up from a terminal node. The algorithm cannot know whether the current cell is the left or right child of its parent. If it is the left child, its tag will appear as the second element of its parent's tuple; if a right child its tag will appear as the third element. Backing up is therefore a two-step operation. The Backup procedure begins by assuming the current cell is a left child. It performs an un-car operation, and then verifies that the retrieval succeeded. If so, the assumption was correct, and the algorithm can now proceed to examine the cdr of the parent cell. If the retrieval failed, the terminal node must have been a right child rather than a left child. The contents of TAG and CDR spaces are now invalid, but CAR space, which was clamped during the retrieval, still holds the tag of the child. After copying the contents of CAR space into CDR space, a new retrieval can be run with CDR space clamped to find the correct parent.

Whenever it backs up from a node that is a right child, the algorithm performs the backup procedure again. It will continue doing so until it either backs up from a node that is a left child, or it backs up to the root from a right child. In the latter case the entire tree has been visited, so the algorithm terminates.

Figure 6 shows a sample binary tree, and Fig. 7 gives its representation as a set of triples. A complete list of steps the traversal algorithm goes through when applied to this tree is shown in Table 1.

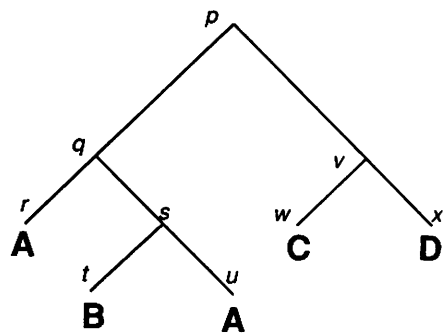

Fig. 6. A binary tree. Tags used for the tuple encoding are shown in italics.

TAG	CAR	CDR
(p	q	v)
(q	r	s)
(r	A	r)
(s	t	u)
(t	B	t)
(u	A	u)
(v	w	x)
(w	C	w)
(x	D	x)

Fig. 7. The encoding of Fig. 6 as a set of triples.

Table 1
Steps in traversing the tree of Fig. 6.

Step	Retrieval cue	Retrieved tuple	Action or comment
Traverse		(p, q, v)	*Remember root is p.*
DownCAR	$(q, \_, \_)$	(q, r, s)	
DownCAR	$(r, \_, \_)$	(r, A, r)	transmit "A"
Backup	$(\_, r, \_)$	(q, r, s)	
DownCDR	$(s, \_, \_)$	(s, t, u)	
DownCAR	$(t, \_, \_)$	(t, B, t)	transmit "B"
Backup	$(\_, t, \_)$	(s, t, u)	
DownCDR	$(u, \_, \_)$	(u, A, u)	transmit "A"
Backup	$(\_, u, \_)$	...	*Associative retrieval failed.*
	$(\_, \_, u)$	(s, t, u)	
Backup	$(\_, s, \_)$	...	*Associative retrieval failed.*
	$(\_, \_, s)$	(q, r, s)	
Backup	$(\_, q, \_)$	(p, q, v)	
DownCDR	$(v, \_, \_)$	(v, w, x)	
DownCAR	$(w, \_, \_)$	(w, C, w)	transmit "C"
Backup	$(\_, w, \_)$	(v, w, x)	
DownCDR	$(x, \_, \_)$	(x, D, x)	transmit "D"
Backup	$(\_, x, \_)$	...	*Associative retrieval failed.*
	$(\_, \_, x)$	(v, w, x)	
Backup	$(\_, v, \_)$		*Associative retrieval failed.*
	$(\_, \_, v)$	(p, q, v)	*At root, so done.*

5.2. General tree manipulation

General trees, in which interior nodes are labeled and may have any number of descendants, must be represented indirectly if one is using linked lists. As mentioned previously, interpreted LISP programs are trees of this form. An associative retrieval machine can manipulate general trees the same way LISP does. For example, imagine that the parse tree of Fig. 8 is represented as a linked list. Each node of the tree is a cons cell chain; the car of the first cell holds the node label, and the remaining cells hold the tags of the node's

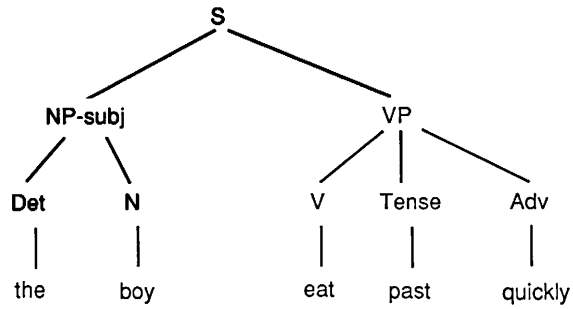

Fig. 8. Parse tree for "The boy ate quickly."

children. We assume that terminal nodes are represented as chains with no children, i.e., in parenthesis notation the tree would be written:

```
(Sent
  (Subject-NP
    (Det (the))
    (N (boy)))
  (VP
    (Verb (eat))
    (Tense (past))
    (Adv (quickly))))
```

The following procedure locates a particular child of a parent node, given the child's label as input. For example, if the current node were VP, Find NamedChild("Tense") would make the second child of the VP node be the current node. We assume that the tuple representing the parent node (i.e., the tuple for the head cell in the parent node's cons cell chain) resides in the Tuple Buffer when the procedure is invoked.

```
FindNamedChild(x) =
   {loop
     GetCDR;              —find next child
     GetCAR;              —fetch child's label
     if $CAR.eql.x then   —if this is the child we want
       exitloop           —then exit
     else
       un-CAR;            —else back up to the parent chain
     endif                —and iterate to check next child
   endloop}
```

If node labels are unique, associative retrieval eliminates the need to search a tree sequentially. For example, we can access any node of Fig. 8 in constant

time with the following procedure:

```
FindNamedNode(x) =
  {$x→CAR;
   $Retrieve.by.CAR}
```

The next procedure finds the parent of the current node by using associative retrieval to follow pointers backward. It assumes that the model is able to distinguish between symbols that are used as atoms and symbols that are used as tags for composite objects. Only symbols denoting atoms can serve as node labels.

```
FindParent =
  {un-CAR;                    —back up to parent's chain
   loop
     un-CDR;                  —back up to previous child
     if $CAR.is.atomic.then   —here's the parent's node label
       exitloop
   endloop}
```

5.3. A richer representation for general trees

We now consider a richer representation for trees that allows access to a node's parent or any of its siblings or descendants with a single associative retrieval. Each node will be a five-tuple:

(tag, label, parent, rsib, lchild) .

The tag field, as before, serves as a unique id for the tuple. The label field contains the node's label. A tree might have several nodes with the same label, but they would have different tags. The parent field holds the tag of the parent of this node. The rsib field holds the tag of the node that is the right sibling of this node. (If a node is a rightmost child, its rsib field will contain the parent's tag.) The lchild field holds the tag of the node's leftmost child, or the node's own tag if it has no children. Figure 9 shows part of a tree represented this way, and Fig. 10 shows the architecture of a hypothetical BoltzCONS network called BoltzCONS-5 for supporting this richer tree representation.

Using this representation, the procedures for finding a node's parent, right sibling, and leftmost child are straightforward associative retrievals similar to the ones we've seen before. Certain other retrievals are a little more complex. To find a node's left sibling, we look for a tuple with the same parent as the current node, and the current node's tag in its rsib field. This search combines two cues into a single associative retrieval by clamping two symbol spaces simultaneously:

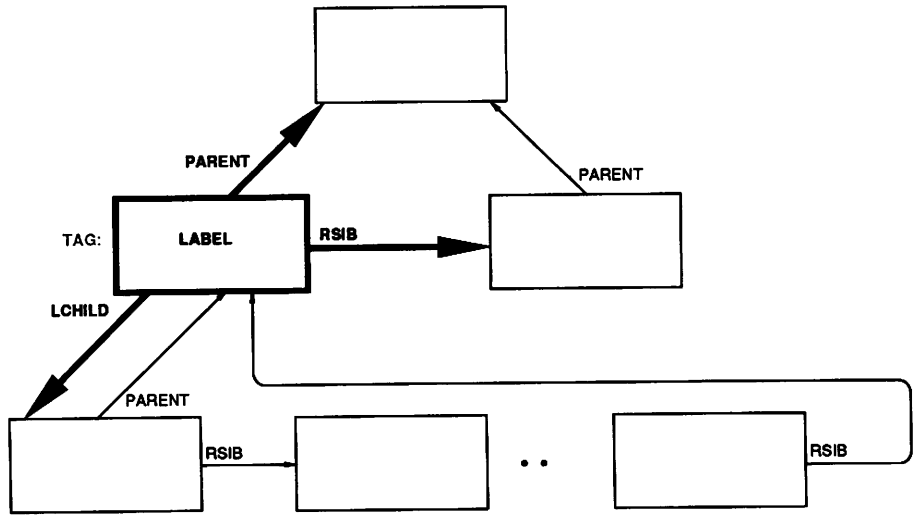

Fig. 9. A richer representation for tree structures.

```
GetLeftMostSib =
  {$TAG→RSIB;
   $Retrieve.by.PARENT.&.RSIB}
```

To find a node's rightmost child we can exploit the fact that sibling chains terminate by pointing back to the parent node. We simply search for a tuple

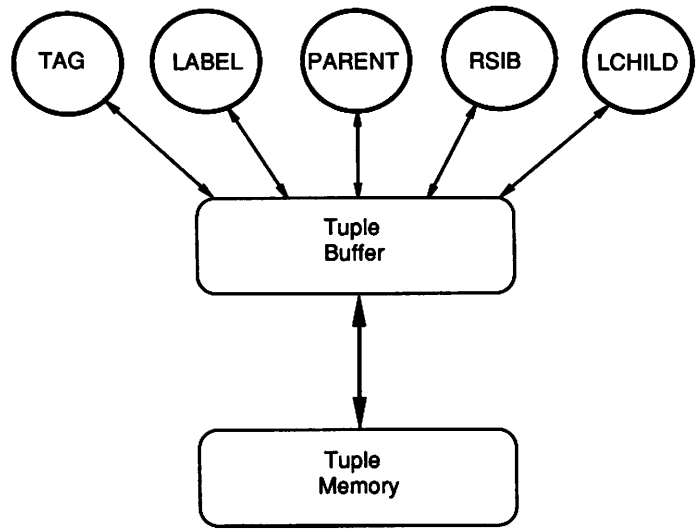

Fig. 10. A version of BoltzCONS that could support richer tree representations.

with this node's tag in both the parent and rsib fields:

```
GetRightMostChild =
  {$TAG→PARENT & $TAG→RSIB;
   $Retrieve.by.PARENT.&.RSIB}
```

To locate an arbitrary named child of the current node, we search for the tuple with this node as parent and the specified label; there is no problem if other nodes in the tree have the same label as long as they have a different parent.

```
GetNamedChild(x) =
  {$TAG→PARENT & $x→LABEL;
   $Retrieve.by.PARENT.&.LABEL}
```

It should be clear from these examples that associative retrieval models are not limited to reproducing the functionality of LISP cons cells. Any computational architecture based on pointers and structured objects is potentially implementable this way.

6. Connectionist Implementation

The low-level organization of BoltzCONS is similar to that of DCPS, Touretzky and Hinton's distributed connectionist production system [24, 28, 29]. It is constructed from essentially the same modules, hooked together in a different way. This section gives an overview of the model's wiring and principles of operation.

6.1. Distributed memory representation

The organization of Tuple Memory is similar to the Working Memory of DCPS. I will describe the simplest version of BoltzCONS with only three symbol spaces. Extension to more elaborate versions is straightforward.

Starting with a 25-symbol alphabet, there are $25^3 = 15{,}625$ triples that might appear in the tuple memory. We assume that the memory is very sparse, so that only a small fraction of these triples, typically one half to two dozen, will be present at any one time. Since the memory is extremely sparse, coarse coding (explained below) can be used to reduce the number of units required while adding a measure of redundancy and fault tolerance to the representation [10, 17].

Tuple Memory consists of 2000 units, each of which has a randomly-generated 6×3 receptive field table such as the one shown in Fig. 11. The table has three columns because we are encoding triples. The choice of six rows is not critical; it yields good performance (as measured by memory capacity, noise immunity, and amount of real memory required by the simulator), but five or seven rows would also work.

C	A	B
F	E	D
M	H	J
Q	K	M
S	T	P
W	Y	R

Fig. 11. An example of a randomly-generated receptive field table for a Tuple Memory unit. The receptive field of the unit is determined by the cross product of the three columns.

The receptive field of a unit is the set of triples generated by the cross-product of the three columns of its receptive field table. A receptive field contains $6 \times 6 \times 6 = 216$ triples, so each receptor covers approximately 1.4% of the space of all possible triples. Receptors are therefore "coarsely tuned"; hence the term "coarse coding." This is also an example of a distributed representation, because triples average $(6/25)^3 \times 2000 = 27.648$ receptors each. Activity in any one receptor does not constitute a representation of any particular triple. Only a collective pattern of activity across a set of receptors corresponds to a triple.

To store a triple in Tuple Memory one turns on all the units in whose receptive field it falls. If we stored the triple (F, A, B), for example, we would turn on the unit depicted in Fig. 11 because it has an F in column 1, an A in column 2, and a B in column 3. We would also turn on roughly 27 other units that also meet these specifications. If we then stored (F, C, D), its activity pattern, which also contains about 28 units, would be superimposed (via inclusive-or) on top of the previous pattern. The result is shown in Fig. 12, in which 55 of the 2000 units are active.

An external observer can tell whether a triple is present by checking the percentage of its receptors that are active. If the percentage is large enough, e.g., at least 75%, the triple may be deemed to be present. BoltzCONS does not actually compute these percentages; the relative activation strengths of triples determine which ones will be found when an associative retrieval is performed. However, for debugging purposes it can be useful to display the percentages of the most highly activated triples. Table 2 provides this information for the case where (F, A, B) and (F, C, D) have just been stored. There is a clear gap between present and absent triples; the strongest triple not actually present has only 40% activation. As the memory fills up, this gap gradually narrows.

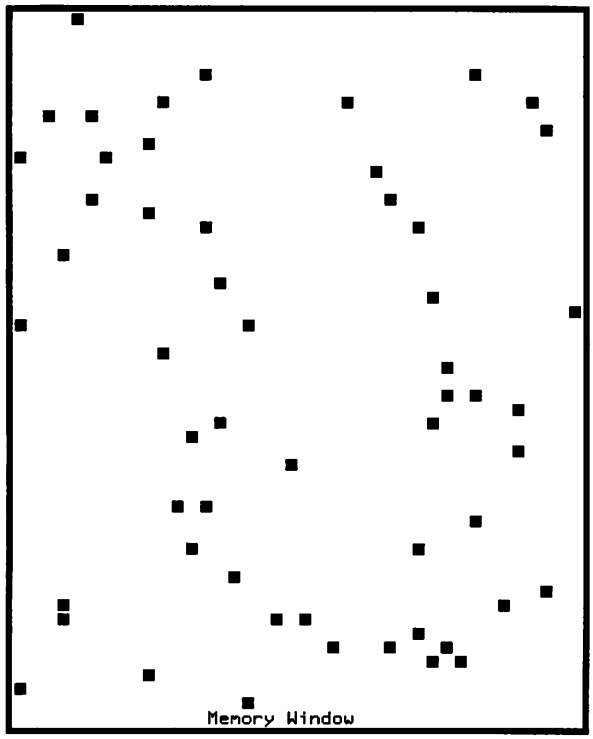

Fig. 12. The state of Tuple Memory after the triples (F, A, B) and (F, C, D) have both been stored. There are 55 units active out of 2000. There is no significance to the positions of these units.

Table 2
The first dozen triples with the strongest representations when (F, A, B) and (F, C, D) have been stored in memory

Triple	Levels of triple activation		
	Percent active	Active receptors	Total receptors
(F, A, B)	100%	28	/28
(F, C, D)	100%	28	/28
(F, A, D)	40%	11	/27
(F, B, D)	38%	10	/26
(F, A, X)	37%	11	/29
(S, A, B)	37%	10	/27
(F, Q, D)	37%	10	/27
(F, C, N)	37%	10	/27
(F, C, B)	37%	10	/27
(F, C, M)	35%	10	/28
(F, T, D)	35%	10	/28
(N, C, D)	34%	10	/29

Figure 13 is a graph of the activation levels of all 15,625 triples after (F, A, B) and (F, C, D) have been stored. The dots in this figure are associated with triples, not with Tuple Memory units. Triple (A, A, A) is in the upper left-hand corner, and (Y, Y, Y) in the lower right. The size of a dot indicates the number of active Tuple Memory units whose receptive field includes that triple. The dark horizontal band in the top quarter of the figure, called the "F-band," is an artifact of our storing two triples that both begin with F. The two darkest spots in this band correspond to (F, A, B) and (F, C, D). A moderate thresholding operator applied to this figure produces Fig. 14, where the triples with the highest activity levels stand out more clearly.

The 216 triples in a unit's receptive field are not chosen independently. They are generated by a Cartesian product of three sets, thereby forming a Cartesian subspace of the entire symbol space. Similar triples will therefore tend to share receptors. This is important for associative retrieval (in particular, it allows the TAG, CAR and CDR spaces to extract the components of a triple), but it can also lead to interference effects if the memory fills up, or if many similar triples are stored. Table 3 shows the expected number of receptors two triples share as a function of the number of components they have in common, c. The

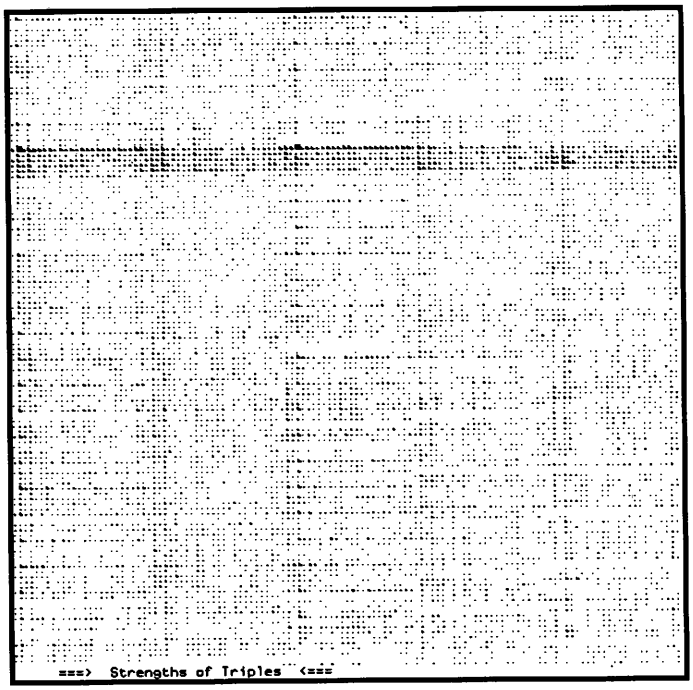

Fig. 13. The levels of support for all 15,625 triples after (F, A, B) and (F, C, D) have been stored in the Tuple Memory, represented by the 55 active units in Fig. 12.

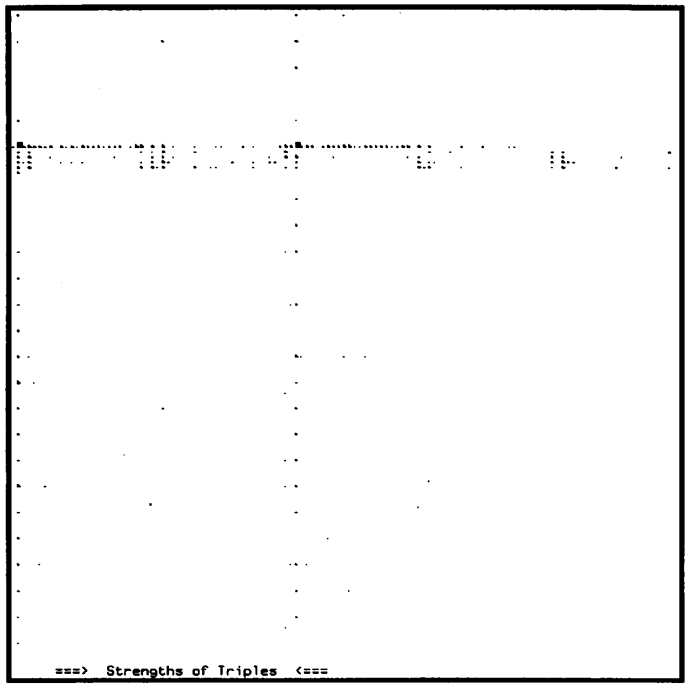

Fig. 14. A moderately thresholded version of Fig. 13, where the (F, A, B) and (F, C, D) spots stand out more clearly.

expected fraction of overlap between receptors of two triples is $(\frac{5}{24})^{3-c}$; the number of shared receptors is therefore $(\frac{5}{24})^{(3-c)} \times (\frac{6}{25})^3 \times 2000$.

When several very similar triples are stored, a phenomenon called "local blurring" results. This is illustrated in Fig. 15. The four triples (F, A, A), (F, A, B), (F, A, C), and (F, A, D) have all been stored in Tuple Memory. Other triples in the same local neighborhood of Cartesian product space, such as (F, A, E), have a moderately high number of active receptors due to the overlapping representation. This makes it difficult to decide whether they are

Table 3
Degree of overlap between similar triples

Numer of symbols in common	Expected number of shared receptors	Expected percent of overlap
0	0.25	0.9%
1	1.20	4.3%
2	5.76	20.8%
3	27.65	100.0%

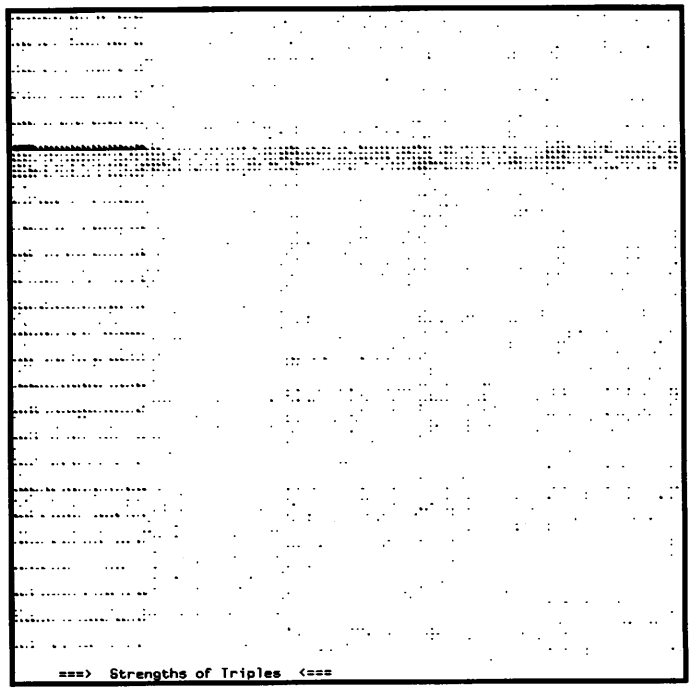

Fig. 15. An illustration of local blurring from storing four closely-related triples: (F, A, A), (F, A, B), (F, A, C), and (F, A, D). Other, similar triples receive a high degree of support, as shown by the dark (F, A, x) line at the beginning of the F-band and the weaker (x, A, y) lines in other bands. A light thresholding operator has been applied to enhance the image.

really present in the memory or not. But it is still clear that unrelated triples, such as (G, K, Q), are absent.

Two other interesting properties of the memory are worth mentioning. First, it has no fixed capacity; it does not "fill up" in the conventional sense. Rather, as more items are added, the gap between present and absent triples narrows. The result is a gradual decrease in retrieval accuracy, as the network finds it increasingly difficult to distinguish triples that were actually stored from those that emerged from overlaps with other triples. This sort of smooth performance degradation, rather than sudden failure when a limit is exceeded, is characteristic of connectionist models.

The second interesting property is the gradual decay of stored triples after a long sequence of deletions of other triples. The more closely related the deleted triples are to the stored one, the faster the fade out effect. This phenomenon is again a consequence of the overlapping representations that form a coarse-coded memory. One way to counteract the decay effect is to recall a triple before it completely fades away. We can then use the TAG,

CAR, and CDR spaces to regenerate the complete pattern for the triple in the Tuple Buffer, and store the completed pattern back into Tuple Memory.

A recent study of the mathematics of this coarse coded symbol representation indicates that it scales well and permits smooth tradeoffs among memory capacity (the number of items simultaneously representable), alphabet size, and accuracy of retrieval [17].

6.2. The Tuple Buffer

The Tuple Buffer serves two distinct purposes. It is used to associatively retrieve individual tuples from the Tuple Memory, given a cue from one of the TAG, CAR, or CDR spaces. It is also used to assemble new tuples by combining TAG, CAR, and CDR inputs. This section concentrates on just retrieval.

During retrievals, the Tuple Buffer acts as a *pullout network* that extracts one member from a collection of superimposed patterns, given some partial specification of the pattern desired. The term "pullout network" is due to Michael Mozer, who invented the concept independently at the same time as I was implementing it under the name "clause space" in DCPS. Mozer proposed the pullout network as a means for a perceptual system to attend to one object in a complex scene [13]. In DCPS, the two clause spaces extract elements from working memory such that together the two elements match the left-hand side of one production rule, and also satisfy a variable binding constraint common to all rules.

The three components of a pullout network as used in DCPS and BoltzCONS are: a one-one excitatory mapping between the units in some distributed memory and the units of the pullout network; a competitive or lateral inhibition mechanism that limits the total amount of activity in the pullout network to roughly enough to represent a single item; and finally, a set of excitatory biases from higher-level spaces that determine which item the network should pull out from memory. These components are illustrated in Fig. 16.

The Tuple Buffer of BoltzCONS consists of 2000 units, with one-one excitatory connections from the 2000 Tuple Memory units. The Tuple Buffer units have very high thresholds, counterbalanced by strong positive weights from Tuple Memory. The result is that no matter how much excitation a Tuple Buffer unit receives from the symbol spaces, it will not become active unless its corresponding Tuple Memory unit is also active. This assures that the Tuple Buffer can only retrieve existing tuples; it cannot hallucinate nonexistent ones. Top-down excitatory biases are supplied by units in the TAG, CAR, or CDR spaces, depending on which cue we are using for the retrieval.

The regulatory unit in Fig. 16 provides the lateral inhibition required by the pullout network. It receives excitatory inputs from all Tuple Buffer units; its

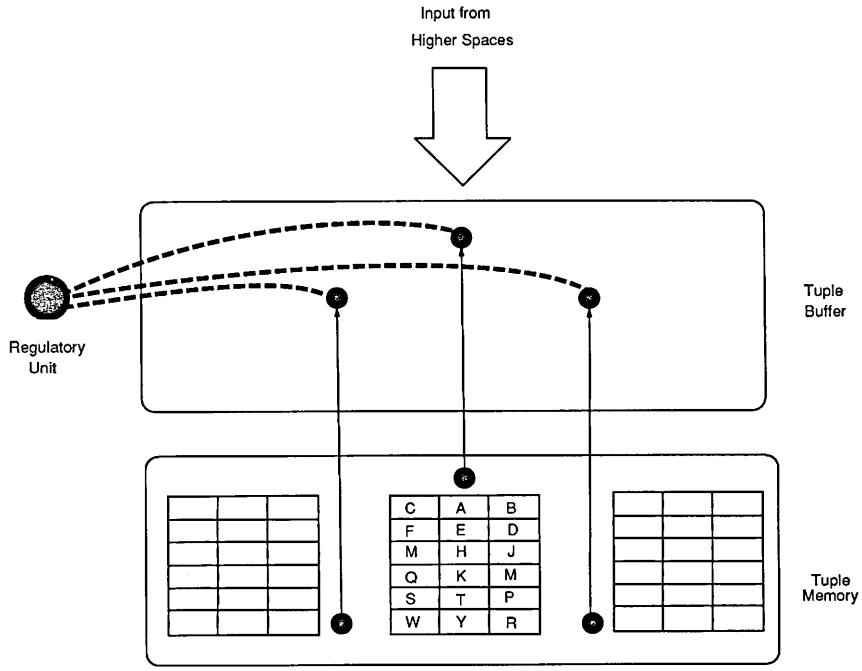

Fig. 16. The structure of the Tuple Buffer acting as a pullout network.

graded output is then fed back to the Tuple Buffer units via inhibitory connections. The amount of inhibition is set so that only about 28 units can remain active in the Tuple Buffer, which is just enough to represent one tuple. Exactly which tuple is chosen depends on the top-down biases the Tuple buffer receives from the symbol spaces.

The use of a regulatory unit with $2N$ asymmetric connections (N excitatory inputs and N inhibitory outputs, where N is the size of the Tuple Buffer) and a graded rather than binary response would appear to violate the definition of a Boltzmann machine. However, this structure is shown in [29] to be equivalent to a pure Boltzmann machine in which the regulatory unit is replaced by $\frac{1}{2}N^2$ bidirectional inhibitory links between pairs of Tuple Buffer units. The advantage of using a regulatory unit is that it implements lateral inhibition more efficiently. Similar regulatory functions have been ascribed to interneurons in real nervous systems.

6.3. Symbol spaces

The TAG, CAR, and CDR spaces are called symbol spaces because their global activity patterns represent individual symbols rather than tuples of symbols. Each space is organized as a coarse-coded, distributed winner-take-all

network containing 25 cliques, one for each of the 25 symbols in the alphabet. See Feldman and Ballard [4] for a description of winner-take-all networks. Each clique has 72 units that vote for its symbol. Each unit, being coarse-coded, votes for three symbols. Thus, the symbol space contains $\frac{1}{3}(25 \times 72) = 600$ units.

The units within a clique support each other via excitatory connections, while units in rival cliques compete with each other via inhibitory connections, as shown in Fig. 17. (If two units have at least one symbol in common, the connection is excitatory.) The stable states of a symbol space are those where all the units in one clique are active, and all the units in rival cliques are inactive. Each of these stable states constitutes a global energy minimum when the symbol space is considered in isolation. In actuality, connections from units in other spaces bias the symbol space units so that one of the 25 stable states will become a deeper energy minimum than any of the others.

Just as the Tuple Buffer is used both for retrieving tuples and for creating new ones, the symbol spaces also play multiple roles. When a symbol space is clamped, its units are prevented from changing state, but those that are active supply top-down input to Tuple Buffer units via weighted connections between the two spaces. For example, suppose the symbol *F* is clamped into TAG space. During an associative retrieval, TAG units will supply excitatory inputs to those Tuple Buffer units having *F* in the first column of their receptive field tables. Each active Tuple Memory unit tries to turn on its corresponding Tuple Buffer unit, but lateral inhibition limits the number of active Tuple Buffer units to about 28. The bias supplied by the active TAG units will cause the Tuple Buffer to select only the active units from Tuple Memory whose receptive field tables contain *F* in the first column. Thus, given the retrieval cue (*F*, _, _), the Tuple Buffer selects some tuple beginning with *F*.

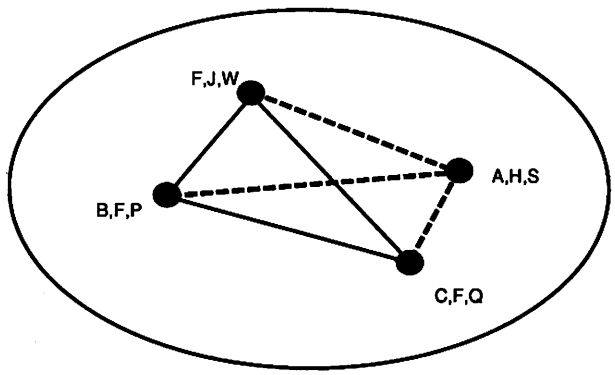

Fig. 17. Four nodes out of the 600 that make up a symbol space. Three of the nodes excite each other because they include *F* in their receptive field; they inhibit the fourth node, which does not vote for *F*. (Inhibitory connections are drawn as dashed lines.)

The CAR and CDR spaces are also connected to the Tuple Buffer. Suppose the tuple being retrieved from memory is (F, A, B). The units active in the Tuple Buffer will have an A in the second column of their receptive field table, and a B in the third column. Since the CAR and CDR spaces are not clamped, they are free to change state during the retrieval, and they are influenced by the pattern emerging in the Tuple Buffer. This will eventually cause the A clique to win the competition in CAR space. Its stable state has a deeper energy minimum than the other letters, due to the bias supplied by the pattern in the Tuple Buffer. Similarly, the B clique will win in CDR space.

The connections between symbol spaces and the Tuple Buffer are bidirectional, which means that even the unclamped symbol spaces will influence the behavior of Tuple Buffer units. During a retrieval with the cue $(F, \_, \_)$, if there are several items stored in memory, there will probably be more than just 28 active units that happen to have an F in their receptive field table, even if only one stored triple begins with F. The collective action of the CAR and CDR spaces helps the Tuple Buffer to home in on the roughly 28 active units that collectively code for a single tuple, such as (F, A, B), that all 28 units support.

6.4. Creating and deleting tuples

New tuples are created by clearing the Tuple Buffer and clamping the component symbols into their respective symbol spaces. The units in each symbol space will excite those Tuple Buffer units to which they have connections. The thresholds of the Tuple Buffer units can be manipulated (by shutting off input from Tuple Memory and supplying a nonspecific bias signal to all units in the buffer) so that only those tuple buffer units that receive excitation from the units in all three symbol spaces will become active. In effect, when creating a new tuple the buffer computes the intersection of the activation it receives from the TAG, CAR, and CDR units, as shown in Fig. 18.

After a new activity pattern has been established in the Tuple Buffer, it is copied into the Tuple Memory by the mechanism shown in the bottom half of Fig. 18. A set of gated one-one connections between the buffer and the memory allow each active Tuple Buffer unit to turn on its corresponding Tuple Memory unit. (Gating is implemented by multiplicative connections, drawn as triangles in Fig. 18.) The gate keeps the connection inactive most of the time, except when a store signal issued. At that time the activity pattern in the Tuple Buffer is transmitted to the Tuple Memory, where it is superimposed onto (that is, inclusive-ored with) any existing pattern there.

Multiplicative connections are not part of the normal Boltzmann machine model, although they are available in the "higher-order" Boltzmann machines defined by Sejnowski [19]. We need not be concerned with that here, however, because during an annealing, which is the only time we rely on the Boltzmann

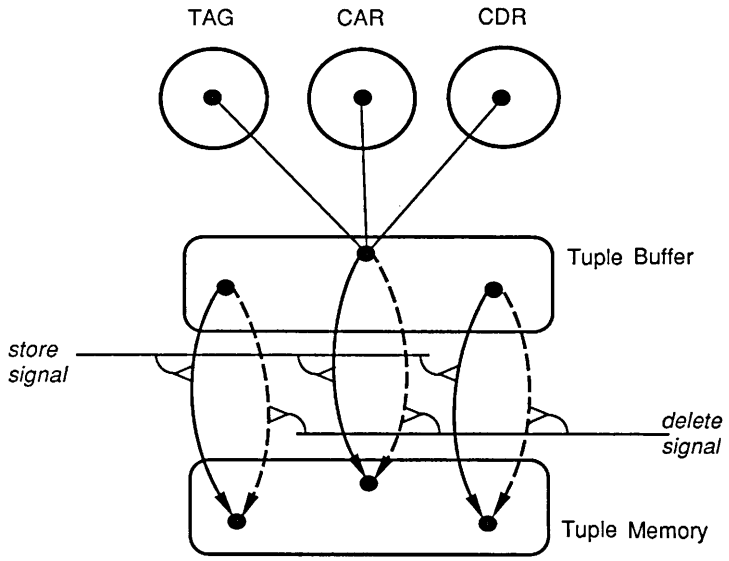

Fig. 18. Connections for storing/deleting the contents of the Tuple Buffer in Tuple Memory.

machine's properties, all gates are fixed. If the gate is open, a gated connection behaves like an ordinary connection; if the gate is closed, the model behaves as if the connection didn't exist.

Tuples are deleted from the Tuple Memory using a set of gated inhibitory connections, also shown in Fig. 18. First, the activity pattern of the tuple to be deleted must be set up in the Tuple Buffer. This is normally done by a retrieval, but it could also be done by assembling the tuple from individual components in the three symbol spaces. When the deletion gate is opened, each active Tuple Buffer unit turns off its corresponding Tuple Memory unit.

The Tuple Memory units do no processing on their own. They serve merely as latches to hold an activity pattern and apply it as input to the Tuple Buffer. The same effect could be achieved without a Tuple Memory if the Tuple Buffer units had modifiable rather than fixed thresholds.[5] Storing a pattern would be achieved by lowering the thresholds of selected Tuple Buffer units, making them more likely to come on.

On the other hand, storing tuples as activity patterns rather than as modified thresholds allows multiple parallel access to Tuple Memory, as shown in Fig. 19. Some BoltzCONS algorithms, such as those for permuting nodes in a tree, have faster and more straightforward implementations in a multiple-buffer architecture. For example, using the richer tree representation of Section 5.3, the algorithm below inserts a new node into a tree as a right sibling of the node

[5] This observation is due to Hinton.

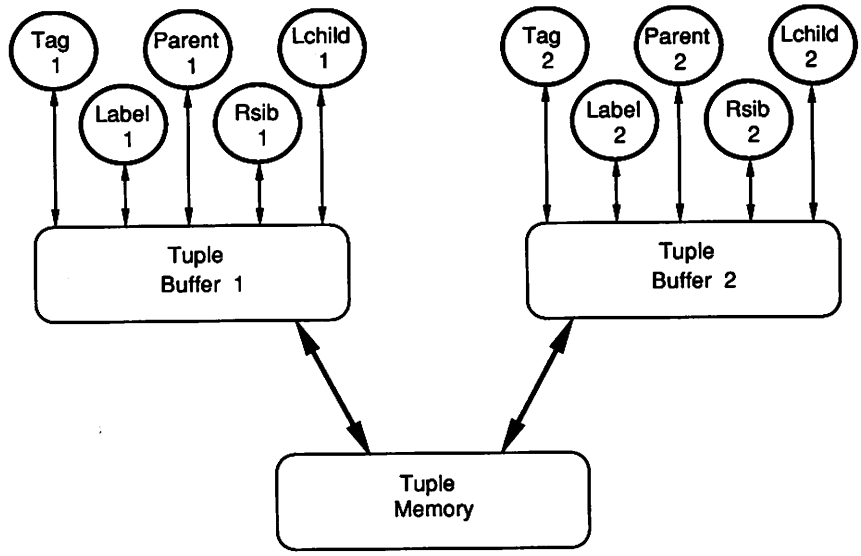

Fig. 19. A variant of BoltzCONS with multiple tuple buffers accessing the same memory.

currently in Tuple Buffer 1. We assume that the components of this tuple are also represented in the TAG1, LABEL1, PARENT1, RSIB1, and LCHILD1 spaces. The new node initially has no children:

```
InsertNode(x) =
  {$NewTag→TAG2 &
     $x→LABEL2 &
     $PARENT1→PARENT2 &
     $RSIB1→RSIB2 &
     $Delete.tuple.1.from.memory;
   $TAG2→RSIB1 & $TAG2→LCHILD2;
   $Assemble.tuple.in.buffer.1 &
   $Assemble.tuple.in.buffer.2;
   $Store.tuple.1 & $Store.tuple.2}
```

6.5. Communication between symbol spaces

Following pointer chains requires copying the activity pattern representing a symbol from CAR or CDR space into TAG space. Other operations, such as following pointers backward, require copying in the opposite direction. Such transfers are trivial if each of the 600 symbol units in the destination space has the same three-letter receptive field as its counterpart in the source space. But even if this is not the case, the transfer of symbols between spaces is easily (and

more robustly) achieved using many-to-many connections between units that have at least one letter in common. In this case the two spaces need not even be of the same size. The transfer method is illustrated in Fig. 20. Suppose the symbol *J* is represented in CAR space, meaning the first, second, and fourth units are active. To transfer this symbol to TAG space we turn off all the TAG units, hold the CAR units clamped so they cannot change state, and open the gate on the two-way connections between the two spaces. The first two TAG units in Fig. 20 will receive excitation from three connections each. The last TAG unit, which does not code for *J*, will receive excitation from only one connection; this spurious excitation comes from the fact that the last units in TAG and CAR space both happen to code for *P*. With a high enough threshold, the units that become active in TAG space will be only those that code for *J*. (Recall that the connections within TAG space will cause *J* units to excite their siblings and inhibit rival units, so any spurious activation from CAR units will be more than compensated for by intra-space lateral inhibition.)

The gate that controls this transfer is actually implemented by a set of multiplicative connections from a line carrying the "transfer" signal onto each of the connections between the two spaces. During an associative retrieval there is no transfer signal (i.e., the gate is closed), so these connections have no effect.

6.6. Associative retrieval by parallel constraint satisfaction

Associative retrieval in BoltzCONS is accomplished by parallel constraint satisfaction using the Boltzmann Machine simulated annealing algorithm. Assume that Tuple Memory holds the set of tuples shown in Fig. 4, and we wish to find the tuple whose tag is ***p***. We begin by clamping the symbol ***p*** into TAG space, that is, we turn on all the TAG units that include ***p*** in their receptive field, and turn off the ones that don't include ***p***. The states of the

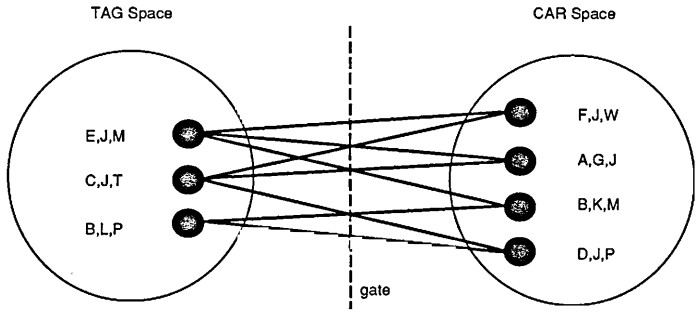

Fig. 20. Gated connections between symbol spaces allow symbols to be copied from one space to another. Using a many-to-many connection pattern, it isn't necessary for units in the two spaces to have identical receptive fields; the two spaces need not even be of the same size.

TAG units are then frozen so that they cannot change during the annealing. As the annealing begins, the active TAG units supply excitation to the Tuple Buffer units to which they are wired. At the same time, the active Tuple Memory units, representing the superimposed patterns of all eleven stored tuples, are exciting their corresponding Tuple Buffer units. Lateral inhibition in the Tuple Buffer prevents more than a few dozen Tuple Buffer units from being on simultaneously. The units that are most likely to be on are the ones receiving the highest activation, that is, the ones receiving input from Tuple Memory *plus* an extra boost from the clamped TAG space units. Therefore, the Tuple Buffer will tend to choose units that encode a tuple from the Tuple Memory that begins with ***p***.

The CAR and CDR spaces, being unclamped, are free to wander about looking for an energy minimum. As the pattern in the Tuple Buffer develops toward the representation of one particular tuple, it exerts an influence on the CAR and CDR units. In CDR space, for example, units that vote for ***q*** will be getting a lot of excitatory input from active Tuple Buffer units representing (***p***, **Event7**, ***q***). This tends to make the clique for ***q*** be the winner in CDR space.

Because BoltzCONS was built from the components of DCPS, it uses the same settling algorithm for associative retrieval. This algorithm does most of its work very quickly at a single temperature. It is more like rapid stochastic search than simulated annealing.

DCPS solves a much harder problem than BoltzCONS. DCPS must simultaneously retrieve two triples that jointly satisfy two independent constraints: the left-hand side pattern matching constraint and the variable binding constraint. See [24] for details. BoltzCONS, on the other hand, only retrieves one triple at a time, subject to a single constraint: the influence that the clamped symbol space exerts on the Tuple Buffer. Given the simplicity of the retrieval problem in BoltzCONS, it is likely that a simpler settling algorithm would give equally good results. In particular, the Hopfield and Tank model [11], which uses deterministic continuous-valued units instead of stochastic binary ones, seems an attractive alternative.

7. Managing a Distributed Memory

BoltzCONS might be viewed as a short term or working memory model for processing conceptual structures. Human short-term memory has a distinctly limited capacity. The few items present at any given time will quickly fade or be displaced unless some action is taken to retain them. In normal cognitive processing we may expect these items to be created, manipulated, and discarded at a rapid rate.

An acknowledgement of limited memory capacity leads to the question of how memory resources might be allocated and recovered. Too little is known about human information processing to support much speculation on this topic.

However, to pursue the basic pointer structure analogy that gave birth to BoltzCONS, I will sketch and compare some candidate mechanisms.

Let us define *allocation* as the process of finding a fresh tag, meaning a tag not in use by any currently-stored tuple. We define *reclamation* to mean deleting "garbage" tuples representing cells no longer pointed to by any of the concept structures currently in memory. To make this last idea more concrete—without making any psychological claims—we assume that working memory is organized as a tree whose root has a distinguished tag, and whose children, which are unordered, are individual conceptual structures. A structure can be converted to garbage by severing its connection to the root. Smaller bits of garbage might be generated as a side effect of performing minor surgery on graphs, such as deleting or permuting a few nodes. Thus, we expect that normal operations on symbol structures will continually produce garbage tuples that must be reclaimed to prevent the memory from filling up.

7.1. The allocation problem

Here are five increasingly sophisticated schemes for attacking the allocation problem:

(1) *Maintain a freelist*
This technique was used in early LISP implementations: all unallocated cells are strung together in a linked list. A pointer to the head of the list is maintained in a special register. New cells are allocated by popping them off the free list; in BoltzCONS this means deleting the tuple from Tuple Memory and then reusing its tag. The pop operation could be done efficiently in BoltzCONS without interfering with computations in progress by using a second Tuple Buffer with associated symbol spaces, as in Fig. 19. One problem with free lists is their inherent sequentiality. If there are multiple tuple buffers interacting with the Tuple Memory simultaneously to build new structures, maintenance of the freelist becomes a bottleneck.

(2) *Use a special marker for free cells*
Each unallocated cons cell could have a special marker in its car and cdr which could be picked up by associative retrieval. The problem with this approach is that if there are many free cells, local blurring will degrade the accuracy of the memory.

(3) *Mark free cells by setting their car and cdr fields to their tag*
This form of marking avoids local blurring. A constrained form of associative retrieval could be used to find such self-referential cells. However, this would interfere with some of the reclamation schemes proposed below.

(4) *Pick a tag at random and verify that it's unused*
We can use associative retrieval to verify that there is no stored tuple with the

given tag. If the memory is sparse, the first tag we pick will probably be free. If a tag is already in use, we can pick another tag at random and repeat the process. As memory fills up this scheme becomes less efficient. Processing time not only increases, it may also vary substantially depending on how lucky our random guesses turn out to be.

(5) *Use an inhibitory winner-take-all space*
This idea was suggested by David Chapman at the Massachusetts Institute of Technology. Create a special winner-take-all space whose units have inhibitory connections directly from the Tuple Memory; the connectivity is based on the first column of the receptive field table. The winner-take-all units will have negative thresholds (i.e., positive biases) which cause them to turn on in the absence of inhibition from Tuple Memory. The stable states of this network will then correspond to symbols that do not appear as the tag of any tuple present in Tuple Memory. We will also need excitatory connections within each clique of units and inhibitory connections between rival cliques, as in an ordinary winner-take-all space, to assure that only one symbol at a time is chosen as the winner.

7.2. The reclamation problem

Here are four schemes for removing tuples that are no longer accessible.

(1) *Activation decay of units in Tuple Memory*
If the activation levels of tuple memory units are made to decay slowly towards zero, the tuples those units represent will gradually fade from the memory. We can retain those tuples that continue to be accessed by having each Tuple Buffer unit refresh the activity level of its corresponding Tuple Memory unit at the end of every associative retrieval. One problem with this approach is that, since the components of an object are represented by independent tuples, if we continually access only some parts of an object, the other parts might fade away, leaving an incomplete structure in memory. It would be impractical to traverse an entire tree to refresh its tuples any time we access any part of it. However, in architectures such as Touretzky and Geva's DUCS model [27], where composite objects are represented by a single large activity pattern that is always retrieved as a whole, the use of exponential decay with refresh-on-retrieval is feasible.

It's not really clear if this is the right kind of reclamation, though. If the system gets busy (i.e., is suddenly called upon to do a lot of processing), it will be generating and discarding new structures very rapidly. The decay mechanism might then be unable to reclaim these structures quickly enough to prevent memory from filling up. We could compensate for the higher processing speed by increasing the decay rate of Tuple Memory units, but this will force us to waste more processing resources on refresh. It would be better if garbage were displaced by new data rather than simply allowing garbage to decay.

Another source of decay comes from the effect of overlapping representations due to coarse coding. Even if units' activity levels remain constant, as we delete many triples, some will share units with triples that were not meant to be deleted. After many deletions, a triple can fade from memory unless it has been periodically refreshed.

(2) *Pick a tuple at random and see if no car or cdr points to its tag*
If so, delete it, then check its car. If nothing points to the car, reclaim it and continue along the car path; otherwise check the cdr. When we get to the end of a chain of reclaimed tuples we go back to random search again. If we use a dedicated Tuple Buffer for this purpose, reclamation can occur in parallel with, and completely independent of, the creation and manipulation of non-garbage structures.

(3) *Monitor deletions*
If the storage recovery mechanism is in random search mode when BoltzCONS deletes a tuple, it should immediately check the car and cdr of the deleted tuple to see if they are tags (rather than atoms), and if any tuples other than the one just deleted contain those tags. If not, it can begin following a reclamation path, starting with the subtree beginning with the now orphaned tag, as in case (2). If both the car and cdr point to now-orphaned subtrees, the storage recovery mechanism need only pursue one of them. The other will be picked up later by random search.

(4) *Detect garbage tuples directly*
One could perhaps design a special module to detect garbage tuples, i.e., tuples not pointed to by any other tuple. This would be yet another variant on the familiar winner-take-all network. A unit that votes for, say, the tag J, would have excitatory connections from Tuple Memory units with J in their tag fields, and also inhibitory connections from Tuple Memory units with J in either their car or cdr fields. Such a network should settle into a stable state representing the tag of a tuple that was stored in memory but not pointed to by any other stored tuple. One could then perform an ordinary associative retrieval to access the tuple with this tag and delete it from memory. This wouldn't detect circular garbage, however.

Although use of dedicated hardware might seem an expensive way to detect and reclaim garbage, recall that BoltzCONS is modeling processing in short-term or working memory, not long-term memory. Since the short-term memory contains only a few moderately complex structured objects at a time, and its contents are subject to rapid turnover, special storage recovery circuitry might not be unreasonable.

8. Discussion

In the course of this paper, BoltzCONS evolved from a parallel associative

implementation of LISP cons cells into a more powerful and general purpose symbol processor. The full architecture supports direct representations of arbitrary tree structures based on a 5-tuple encoding, and it can perform complex pointer manipulations using multiple buffers operating simultaneously on its tuple memory. A variety of methods for dynamic allocation and reclamation of memory resources were also discussed. In this final section I will try to put the BoltzCONS model in perspective.

8.1. BoltzCONS and implementational connectionism

BoltzCONS joins many other connectionist models in addressing an important implementation question: How can intelligence emerge from the collective activity of a mass of neurons? One might approach this question at many levels. At an extremely abstract theory of computation level, idealized neurons function as boolean logic gates and latches, from which one can wire up a Turing machine or any digital computer. Of course this approach ignores many crucial biological constraints, such as the fact that individual neurons are unreliable devices, or the fact that the human genome does not contain enough information to specify precise point-to-point neural wiring for something as complex as a computer. At the other extreme, if fidelity to biology rather than theoretical simplicity is the main goal, one could try mapping proposed symbol processing architectures directly onto actual brain structures. Neuroscientists have had great success explaining the early stages of vision in terms of receptive fields, cortical maps, hypercolumns, and so forth, but this approach seems premature for symbol processing, which involves nonsensory, nongeometric representations.

Connectionist symbol processing theories therefore occupy a middle ground between purely abstract computational architectures and purely data-driven biological models. Connectionist modelers are concerned with computational questions more than biological ones, but they try to work within hailing distance of the biologists. BoltzCONS in particular obeys several important biological constraints. Its units are stochastic, and the failure of any one of them would have no observable effect on the model's behavior. Units have coarsely-tuned receptive fields; they do not encode discrete symbols as a grandmother cell would. And the mappings between modules (e.g., between the symbol spaces and the Tuple Buffer) are rich and highly redundant, as is the case in real neural systems.

I do not envision connectionist symbol processing moving closer to real biological explanations any time soon. There are too many layers of description between psychology and neuroscience where our understanding is extremely limited. Connectionists will have to content themselves with studying the computational properties of various abstract architectures, drawing inspiration from biology where possible. One hopes that at some point in the future these

investigations will provide the necessary language for framing a biological explanation of cognition.

8.2. BoltzCONS and cognitive psychology

Like Anderson's ACT* model [1], BoltzCONS raises questions about mental representations. If there really are symbol structures in the brain, one can legitimately ask about their functional properties, i.e., properties defined at the level of an abstract associative retrieval machine. Suppose, for example, that a particular cognitive theory suggests that people manipulate tree structures in their heads. One can ask: Does the brain use a direct or indirect representation for these trees? What tree manipulation primitives does it provide? If nodes are ordered, is it possible to access the rightmost child of a node as quickly as the leftmost child? How much time does it take to perform various sorts of permutations on nodes? The answers can reveal a lot about the underlying cognitive architecture. BoltzCONS-5 is a good illustration of the way a slight architectural modification can turn a slow serial operation into a fast parallel one.

Of course, people probably have much richer and more complex structures than trees in their heads. The BoltzCONS approach can be extended to develop models of these structures and explore their computational properties. In contrast, if one talks only about manipulating symbol structures, without considering how they could be realized in neuron-like hardware, then it is not meaningful to ask whether certain operations necessarily take longer or require more resources than others.

8.3. Revisionist symbol processing

One of the most pressing questions facing connectionists is how the neuron-like implementation of their models influences their view of what symbol processing is about. If there were no influence, connectionism would merely be an implementation technology rather than an alternative to classical processing. If the influence proves to be fundamental, connectionist modeling could lead to an entirely new understanding of symbol processing. This has not happened yet to any great degree, but one must be patient.

In the case of BoltzCONS, there are several areas where the choice of a connectionist implementation impacts the model. They are: coarse coding, combination of multiple cues, and closest-match search.

Coarse coding is a particularly "neural" representation strategy. The desire to avoid local blurring ruled out certain conventions for marking atoms and the termination of chains in BoltzCONS that would have been perfectly acceptable in noncoarse coded architectures. In addition, coarse coding predicts particular types of error behavior as the memory fills up, and particular types of fading as items are deleted.

The latter two areas where the effects of a connectionist implementation are felt both involve associative retrieval. As mentioned in the introduction, associative retrieval is not unique to connectionist models. But the dividing line between fast and slow operations is drawn differently when one takes a connectionist approach. For example, for a conventional computer to perform the associative retrievals used in BoltzCONS-5, tuples would have to be stored redundantly in five hash tables keyed on the TAG, LABEL, PARENT, RSIB, and LCHILD fields. Given one component of a tuple, it is easy to fetch the entire tuple from the appropriate hash table in constant time, independent of the number of tuples that have been stored. But that is as far as we can push the hash table representation. Other operations that connectionist architectures perform quickly will not map neatly onto hash tables.

One example is the combination of multiple cues, as in Section 5.3, where we used both the PARENT and RSIB fields to constrain a retrieval. In general, supplying more constraints causes a connectionist model to settle faster. But a conventional machine using hash table representations will be slowed down. It will either be forced to intersect the sets of entries retrieved from the PARENT and RSIB hash tables (a serial process), or it will have to maintain additional hash tables keyed on all pairs of fields that might be used in a query. For more complex representations, where the number of ways of combining multiple cues would be much larger, this second approach would not be feasible.

Finally, connectionist associative retrieval architectures do more than retrieve exact matches to a query. If there is no exact match, they can retrieve the closest inexact match. Consider this example from DUCS, a connectionist frame system. Given a frame describing a bird, and a request for the "nose" of the bird, DUCS retrieves the closest slot to "nose," which is "beak." For an elephant frame, the closest matching slot would be "trunk." DUCS encodes slot names as binary feature vectors and uses Hamming distance to determine the closest match. Serial machines cannot access a closest matching item in constant time; hashing works only for exact matches.

In the current version of BoltzCONS symbols are atomic; they have no semantic features that could be usd to determine closest match. But one could easily imagine a version of BoltzCONS in which symbols were semantic feature vectors. For example, Dolan and Smolensky's tensor product production system TPPS (discussed further below), which does allow for semantic features, could be adapted to form TensorCONS, much as DCPS gave rise to BoltzCONS.

8.4. Comparison with other connectionist models

BoltzCONS occupies a unique point in the space of connectionist symbol processing models. Like Pollack's Recursive Auto-Associative Memories [16], it represents objects with fixed numbers of components occurring in fixed

positions. This distinguishes these models from Derthick's μKLONE [2] and Touretzky and Geva's DUCS [27], which can represent objects with variable numbers of components, and in which component names are not tied to fixed positions in a vector. Another common point between BoltzCONS and RAAMs is that pointer following is their central operation. μKLONE performs more interesting sorts of retrievals—actually inferences—based on parallel constraint satisfaction. Following chains of pointers from one concept to another is not its major purpose.

BoltzCONS' primary advantage over μKLONE and RAAMs is that it can create new objects dynamically, by changing the activity pattern in its Tuple Memory. μKLONE's knowledge is fixed in advance by its wiring pattern, which is compiled from a symbolic-level description of the domain. Although it can incorporate a small amount of new information as part of each query, it cannot retain this knowledge from one query to the next, or record the results of its inferences. Hinton's connectionist implementation of semantic nets [7] could acquire new knowledge, but since the semantic net was encoded in weights that were trained by the perceptron convergence procedure, the model could only learn by repeated presentation of the entire set of training instances by some external teacher. When viewed as a short-term or working memory model, its knowledge was static.

RAAMs also have essentially static knowledge, as adding new structure requires laborious training with backpropagation. However, RAAMs have an interesting property: the "pointers" that backprop creates, rather than being meaningless symbols as in BoltzCONS, encode some features of the objects they point to. They can thus serve as what Hinton [8] calls "reduced descriptions" that allow certain inferences to be made directly from the pointers themselves, rather than from the objects the pointers designate.

Smolensky [21] has shown that the coarse-coded representation used in DCPS and BoltzCONS can be viewed as a subset of a tensor product representation, in which each of the component symbols of a triple is replaced by a vector. This is of more than theoretical interest, because it suggests that the pullout mechanism and annealing process might be replaced by a faster inner product operator that collapses a rank-3 tensor to a rank-2 tensor. (The annealing in BoltzCONS is already extremely fast, though; in fact it is more properly termed "quenching.") Dolan and Smolensky [3] recently reimplemented DCPS using tensor product machinery in place of the coarse coded memory, and report encouraging preliminary results. It remains to be seen whether this approach offers any advantage in terms of number of units and connections required, or accuracy of retrieval over the coarse-coded model.

Another way to exploit the tensor product representation would be to make the pointers meaningful instead of arbitrary. In other words, let part of each vector hold an arbitrary identifier, but use the other part to encode salient features of the object being pointed to. This would allow pointers to act as

reduced descriptions, as they do in RAAMs. The hard problem that remains is finding the right encoding of the distal object so that the pointer tells us something useful. (This same problem was encountered in DUCS, which used bit vectors as pointers but did not focus on salient features, since it was domain-independent.) In a RAAM, backprop creates the encoding, but BoltzCONS would have to use another method in order to retain its dynamic qualities. One possibility would be to use backprop to build an encoding network, from general knowledge about a designated domain, that could produce a useful reduced description of any structure in that domain. The encoder could then be wired into BoltzCONS and used to generate meaningful reduced descriptions on the fly.

9. Conclusions

Compared to Rumelhart and McClelland's verb learning model, BoltzCONS does not venture terribly far from traditional notions of symbol processing. Rather than being an eliminative theory, it affirms the reality of the symbolic level. But on the other hand, it is not merely "LISP done with neurons." BoltzCONS shows how the choice of a connectionist implementation can influence one's view of symbol processing in subtle ways. With richer, more complex symbolic representations, the influence of connectionist architectures should be more profound.

In a 1987 paper [26], Mark Derthick and I argued that truly powerful connectionist symbol processors require features of both the BoltzCONS and μKLONE-style models, to combine dynamic flexibility with powerful inference capabilities. Recent progress on many fronts, including work reported in this volume, lends encouragement that this may be feasible.

ACKNOWLEDGEMENT

I thank Geoffrey Hinton, Paul Smolensky, Jordan Pollack, Mark Derthick, Roni Rosenfeld, Charles Dolan, and David Chapman for helpful discussions and suggestions. This work was supported by National Science Foundation grants IST-8516330 and EET-8716324, and by contract number N00014-86-K-0678 from the Office of Naval Research.

REFERENCES

1. J.R. Anderson, *The Architecture of Cognition* (Harvard University Press, Cambridge, MA, 1983).
2. M.A. Derthick, Mundane reasoning by settling on a plausible model, *Artificial Intelligence* **46** (1990) 107–157, this issue.
3. C.P. Dolan and P. Smolensky, Implementing a connectionist production system using tensor products, in: D.S. Touretzky, G.E. Hinton, and T.J. Sejnowski, eds., *Proceedings 1988 Connectionist Models Summer School* (Morgan Kaufmann, Los Altos, CA, 1988) 265–272.
4. J.A. Feldman and D.H Ballard, Connectionist models and their properties, *Cognitive Sci.* **6** (1982) 205–254.

5. J.A. Fodor and Z.W. Pylyshyn, Connectionism and cognitive architecture: A critical analysis, *Cognition* **28** (1988) 3–71.
6. R.D. Greenblatt, T.F. Knight Jr, J. Holloway, D.A. Moon and D.L. Weinreb, The Lisp machine, in: D.R. Barstow, H.E. Shrobe, and E. Sandewall, eds., *Interactive Programming Environments* (McGraw-Hill, New York, 1984).
7. G.E. Hinton, Implementing semantic networks in parallel hardware, in: G.E. Hinton and J.A. Anderson, *Parallel Models of Associative Memory* (Erlbaum, Hillsdale, NJ, 1981).
8. G.E. Hinton, Mapping part–whole hierarchies onto connectionist networks, *Artificial Intelligence* **46** (1990) 47–75, this issue.
9. G.E. Hinton and T.J. Sejnowksi, Learning and relearning in Boltzmann machines, in: D.E. Rumelhart, J.L. McClelland and the PDP Research Group, eds., *Parallel Distributed Processing: Explorations in the Microstructure of Cognition* **1**: *Foundations* (Bradford Books/MIT Press, Cambridge, MA, 1986).
10. G.E. Hinton, J.L. McClelland and D.E. Rumelhart, Distributed representations, in: D.E. Rumelhart, J.L. McClelland and the PDP Research Group, eds., *Parallel Distributed Processing: Explorations in the Microstructure of Cognition* **1**: *Foundations* (Bradford Books/MIT Press, Cambridge, MA, 1986).
11. J.J. Hopfield and D. Tank, "Neural" computation of decisions in optimization problems, *Biol. Cybern.* **52** (1985) 151–152.
12. J.L. McClelland, D.E. Rumelhart and G.E. Hinton, The appeal of parallel distributed processing, in: D.E. Rumelhart, J.L. McClelland and the PDP Research Group, eds., *Parallel Distributed Processing: Explorations in the Microstructure of Cognition* **1**: *Foundations* (Bradford Books/MIT Press, Cambridge, MA, 1986).
13. M.C. Mozer, The perception of multiple objects: A parallel, distributed processing approach, Doctoral Dissertation, University of California, San Diego, CA (1987).
14. A. Newell, Physical symbol systems, *Cognitive Sci.* **4** (1980) 135–183.
15. S. Pinker and A. Prince, On language and connectionism: analysis of a parallel distributed processing model of language acquisition, *Cognition* **28** (1988) 73–193.
16. J. Pollack, Recursive distributed representations, *Artificial Intelligence* **46** (1990) 77–105, this issue.
17. R. Rosenfeld and D.S. Touretzky, Coarse-coded symbol memories and their properties, *Complex Syst.* **2** (4) (1988) 463–484.
18. D.E. Rumelhart and J.L. McClelland, On learning the past tenses of English verbs, in: J.L. McClelland, D.E. Rumelhart and the PDP Research Group, eds., *Parallel Distributed Processing: Explorations in the Microstructure of Cognition* **2**: *Applications* (Bradford Books/MIT Press, Cambridge, MA, 1986).
19. T.J. Sejnowski, Higher-order Boltzmann machines, in: J.S. Denker, ed., *Neural Networks for Computing*, AIP Conference Proceedings **151** (American Institute of Physics, New York, 1986) 398–403.
20. P. Smolensky, On the hypothesis underlying connectionsm, *Behav. Brain Sci.* **11** (1) (1988).
21. P. Smolensky, Tensor product variable binding and the representation of symbolic structures in connectionist systems, *Artificial Intelligence* **46** (1990) 159–216, this issue.
22. D.S. Touretzky, BoltzCONS: Reconciling connectionism with the recursive nature of stacks and trees, in: *Proceedings Eighth Annual Conference of the Cognitive Science Society*, Amherst, MA (1986) 522–530.
23. D.S. Touretzky, Representing and transforming recursive objects in a neural network, or "Trees *do* grow on Boltzmann machines", *Proceedings International Conference on Systems, Man, and Cybernetics*, Atlanta, GA (1986) 12–16.
24. D.S. Touretzky, Analyzing the energy landscapes of distributed winner-take-all networks, in: D.S. Touretzky, ed., *Advances in Neural Information Processing Systems* **1** (Morgan Kaufmann, San Mateo, CA, 1989).

25. D.S. Touretzky, Connectionism and compositional semantics, in: J.A. Barnden and J.B. Pollack, eds., *Advances in Connectionist and Neural Computational Theory* **2**: *High Level Connectionist Models* (Ablex, Norwood, NJ, to appear).
26. D.S. Touretzky and M.A. Derthick, Symbol structures in connectionist networks: Five properties and two architectures, *Digest of Papers*: *COMPCON Spring 87, Thirty-Second IEEE Computer Society International Conference*, San Francisco, CA (1987).
27. D.S. Touretzky and S. Geva, A distributed connectionist representation for concept structures, in: *Proceedings Ninth Annual Conference of the Cognitive Science Society*, Seattle, WA (1987) 155–164.
28. D.S. Touretzky and G.E. Hinton, Symbols among the neurons: details of a connectionist inference architecture, in: *Proceedings IJCAI-85*, Los Angeles, CA (1985) 238–243.
29. D.S. Touretzky and G.E. Hinton, A distributed connectionist production system, *Cognitive Sci.* **12** (3) (1988) 423–466.
30. S. Ullman, Visual routines, in: S. Pinker, ed., *Visual Cognition* (MIT Press, Cambridge, MA, 1984).