

A Graph Neural Network Model of Human and Mammalian Thought

Ken Del Signore
kendelsignore@gmail.com

Abstract: The graph neural network model of human language introduced in (Del Signore 2024) is further developed in several areas via pseudocode. The graph model design closely parallels the Strong Minimalist Thesis design given in (Chomsky, 2019). The Internal and External Merge of the SMT correspond to the left and right modes in the touch() function in the graph model. Workspaces form on every hypernym node as short term memory nodes and Search is done by a linked list of accessible nodes in each workspace. The pseudocode for the Node{} class and touch() function are developed *de novo* using simple language examples. A design objective is to introduce the minimal number of internal variables and/or code changes needed to allow for observed human language phenomena to be replicated. The model is evolved in steps, with each step typically corresponding to a single line of code change. A simple existence proof for the merge based short term memory model is first given as a monte-carlo that simulates a sparsely connected neural network which can form a short term memory association between any two symbols in its lexicon. The model is then developed to examine adjective ordering in noun phrases and hypernym formation in the lexicon. A counting operation is then given with an implementation of a graph network that counts from 0 to 9 forever, or until stopped by the experimenter. A single node with unique parameters is then added to this network that changes the network behaviour to count from 0 to infinity. Further comparisons with the SMT program are then given on the topics of Workspaces and Search.

The neocortex is unique to mammals and gives them their complex behaviour relative to non mammals. Further, across all mammals, the neocortex has the same structure and a ubiquitous internal design (Hawkins 2005). Namely, the neocortex is a two dimensional neural substrate of ~ 5mm thickness that contains layers of neurons clustered into columns with discernable differences between the layers. This structure is common across the entire two dimensional area of the neocortex, and it is common across all mammals. This fact leads to an inference that a single neural algorithm could underlie the operation of the neocortex and that

the algorithm has not evolved any significantly further in any mammals since they first evolved some 225 Mya. The Graph Model, or alternatively the Merge function of the SMT, is posited to represent the symbolic operation of the Mammalian neocortex, including language in post Great Leap humans.

We postulate that a fundamental function of the neocortex is to give Mammals the ability to store any random sequence of symbols from their vocabulary, and then later to be able to recall and make use of the stored sequence. We can shorten a sequence to two symbols, which reduces to the ability to associate any two random lexicon symbols together. The graph model presented herein will introduce the algorithm of creating a new short term memory (stm) node every time a symbol is input. Recent (stm) nodes are associated together through a similar merge mechanism. These (stm) nodes are then processed into long term memory (ltm) nodes during a sleep cycle and the (stm) nodes are cleared.

There are other types of neural network designs built to associate symbols together, such as Hopfield, Bidirectional associative memory, and Associative Neural Networks. All of these networks can map one pattern to another, but it is done with offline training. In the graph model, a new short term memory node is created at run time on symbol input via a merge mechanism; a second order merge mechanism can then associate other recent short term memory nodes together. There is no offline training mode as in other forms of associative memories. However there is a Hebbian process that runs at input time to increase the binding between short term memory nodes under certain conditions; similar to short term plasticity.

We postulate that the Graph Model is a computational implementation of the Strong Minimalist Thesis program (Chomsky et al. 2023). The workspace of the SMT model corresponds to the short term memory nodes in the graph model. The lexicon nodes of the SMT model are the

first level hyponym nodes of the IP node in the graph model, i.e. the interface nodes. In the SMT model, the lexicon nodes create an inscription in the workspace upon input, (Chomsky et al. 2023, fig. 2). In the graph model the inscriptions are the short term memory nodes that are created at input with an Internal Merge between the IP node and the interface node that touched it (Del Signore 2024. fig. 4). Further comparisons with the SMT model are given at the end of the paper.

We define a symbol as anything or any stimulus that the neocortex can conceptualize, such as a word, a sight, or a feeling. Cell Assembly theory posits that groups of neurons are associated together and fire simultaneously to represent cognitive entities (Buzsáki 2011) (Huyck, et al. 2013). In the graph model, the function of a hypothetical cell assembly is encapsulated into a single node as shown in figure 1. Biologically, a symbol is assumed to be an attractor state of some number of neurons that forms on input within the neocortex neural network. If the attractor state is dynamic, the firing rate of the attractor can represent a stored value. The phase of the spike train allows for a negative or positive value

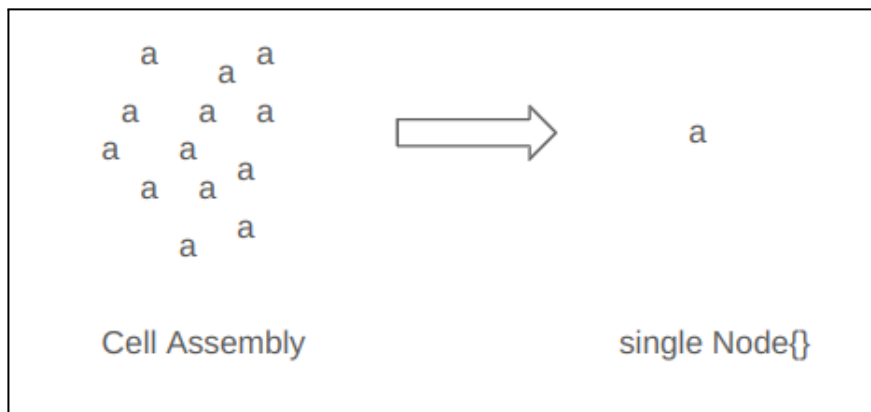


Figure 1 Properties of the cell assembly can be encoded into a single Node instance. The internal variables defined in the Node{} class implies this same information would be carried in a biological cell assembly.

Potential cell assemblies for “thing” and “person/place” have been identified in (Huth et al. 2013), where voxels of size 2mm^3 have been identified in neocortex flatmaps of volunteers that are read stories during fMRI sessions. Using the WordNet hypernym-hyponym database, it is observed that when words that are hyponyms of thing and person/place are read in a story, that two specific voxels can be identified in the same general region across multiple volunteers.

If we consider a sequence of only two symbols, we can define the requirement that an idealized neural network model of the neocortex would have to be able to associate any two symbols together from its vocabulary of symbols and to be able to recall the stored association of the symbols.

The graph model proposes that this functionality can be achieved if the nodes in a network that represent each symbol are sparsely connected to a much larger set of short term memory (stm) nodes (Hawkins 2005). An association is formed between two symbol nodes when they each fire and touch their respective (stm) nodes. If there is a subset of (stm) nodes common to both symbols, these can fire on the combined input and form a short term association between the two symbols. Nodes that have previously fired are assumed to retain this state information in the form of an elevated internal input level such that they will re-fire on a single input from one of the two original sequence nodes.

To demonstrate the association mechanism, we define a simple neural network that contains three symbol nodes, a, b, and c, and some number n of short term memory nodes, denoted () in figure 2. The () nodes are assumed to be randomly connected to the symbol nodes. All links

are bidirectional with a common set of thresholds and weights as shown. A test of the network's ability to associate two symbols would be to input the sequence: | a | b | and then to input | a | again and then the network should output the symbol | b | and not | c |.

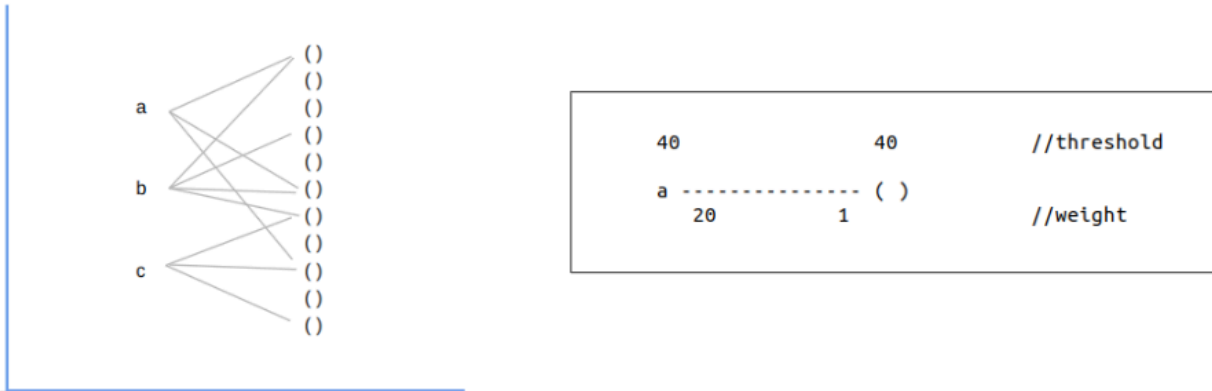


Figure 2: A simple association network. All links are bidirectional. All thresholds are set at 40. Weights are set at 20 and 1 at the symbol and (stm) nodes as shown. These values are chosen arbitrarily with the design goal of having two symbol nodes be capable of firing an (stm) node and a much larger number of (stm) nodes to fire a single symbol node. An association can form between two symbol nodes when they share a sufficient number of (stm) nodes. If the symbol to (stm) connections are not sparse enough, other symbol nodes can fire incorrectly.

The simplified Node{} class is given in figure 2a. Nodes are declared in main() and then accessed by a single function, touch(). If a node fires, the touch() function recursively calls the touch() function for all of the node's branch nodes.

The network in figure 2 can be parameterized by two parameters: n , the number of (stm) nodes and p , the probability that a symbol node will be connected to an (stm) node. For each point in this parameter space, we initialize 100 random networks and run the test cases in table 1 and record how many tests passed and failed at each point (Del Signore 2025). The results are shown in figure 3.

```
Class Node
{
    string          symbol

    int             input_level
    int             threshold

    list<weight, Node*> branches

    touch(...)
}
```

```
touch(...)
{
    if node fires:
        input_level = threshold - 5
        print symbol
        loop over branches{
            branch->touch(...)
        }
}
```

Fig 2a. Pseudocode of the Node class and touch function. After a node fires, its input_level is set to threshold - 5. This choice is made based on the other network parameters in order to cause the association mechanism to occur.

a b a b	-Pass
a b a c	-Fail
a b a no output	-Fail

Table 1: test cases used to tabulate data in figure 3.

This simple monte-carlo demonstrates the capability to associate two symbols using a sparsely and randomly connected cell assembly neural network. The model can easily be expanded to test for larger numbers of symbols by code modification. Increasing the number of nodes still results in a ridge of 98-100% success rate, however the slope of the contour above the ridge becomes steeper.

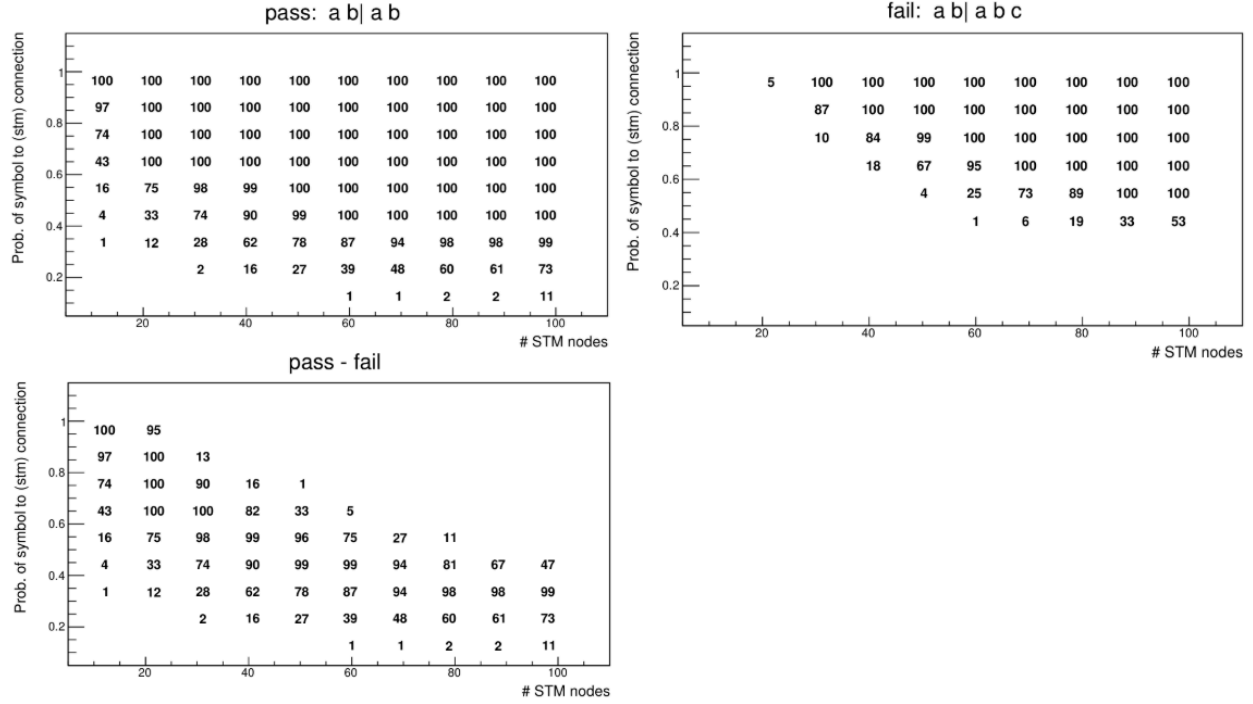


Figure 3 Probing the parameter space of the (stm) demonstration network. Each cell of the scatterplots contains the number of tests (out of 100) that passed or failed (upper two plots). The lower left scatterplot shows the (pass - fail) distribution and shows a ridge in the parameter space with 98-100% model success rate.

In the above monte-carlo, the lexicon consists of three symbols. Each point in the scatterplot represents a ‘forest’ of 100 by randomly generated neural networks¹. The pass and fail tests for the merge operation are run on these random forests and the statistics accumulated into two sets, pass and fail, as shown in figure 3. The existence proof of the merge operation is then obtained by taking the difference between the two sets².

¹ Since nodes are not connected with full merges, but rather with a random probability, the neural networks at each point could be called a forest of vines.

² As in (Marcolli et al. 2025), a disjoint union of two sets of forests of trees is used to demonstrate properties of merge; in this example the disjoint union reduces to a difference of two sets because one set is contained in the other.

We then presume the same encapsulation as before can occur to the (stm) cell assembly, reducing it to a single node as shown in figure 4.

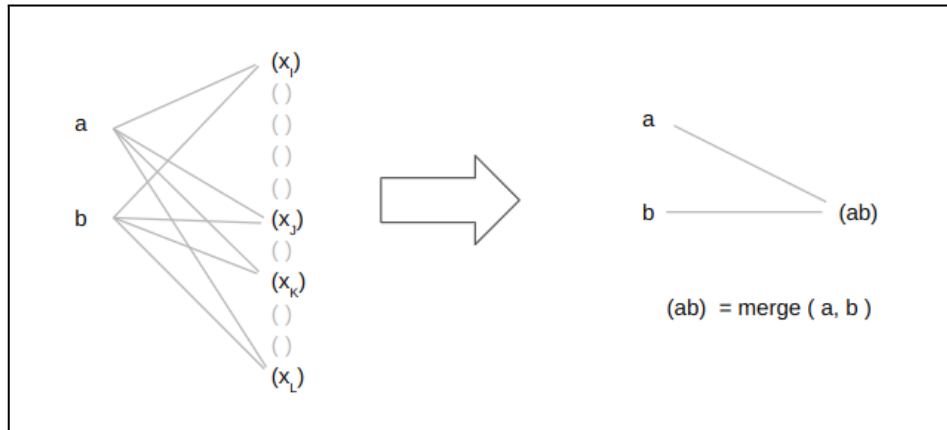


Figure 4 The (stm) cell assembly is encapsulated into a single node (ab). This node is created with a merge(head, copy) function. We assume the (ab) node can inherit any information available from either parent node. We can deduce which information is inherited from which parent from the graph dynamics and the language examples under study.

A Node constructor function is defined based on a merge(head,copy) function. The merge constructor creates a new node that inherits from the head and copy nodes. Two additional links are added to the Node{} class to store the head and copy links as shown in figure 5.

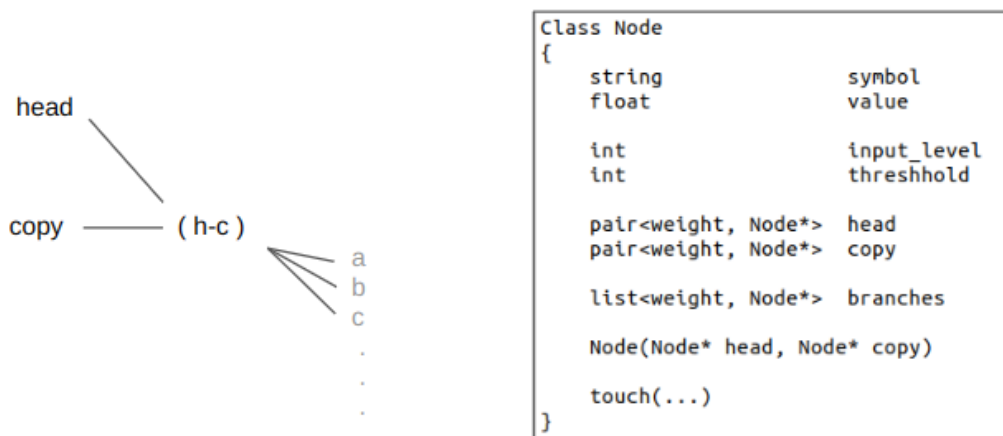


Figure 5. A modified bi-directional Markov graph in which each node has a link to a head and a copy node.

Nodes that are declared in main() are defined as interface nodes. The base node of the graph is labeled “ip” for interface parcel and is declared in main with null pointers to its head and copy nodes. The ip node will have the externalizations [no, ?, yes], corresponding to its internal value = [-1, 0, 1]. Interface nodes are declared with the ip node as their head node and copy node = null. The touch() function will use the null state of the head/copy pointers to detect that it is a surface node in order to externalize a symbol.

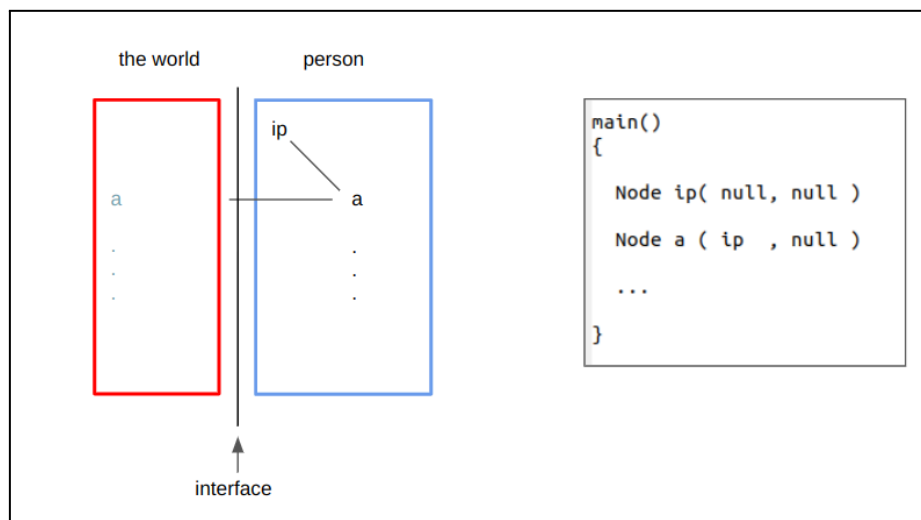


Figure 6 The interface nodes are declared manually in main() as shown.

Interface nodes are touched from main() with touched_by = null as shown in figure 7. In this example the touched interface node “a” will fire and then touch its head node ip; the ip node then fires and will do a merge with the node that touched it, and a new (stm) node will be formed. This (stm) node is shown in figure 7 as the (a) node. After the (a) node is created, the ip node touches and fires it via the recursive (stm)->touch() function call shown in figure 7. The order of node firing is shown in the green box. Input/output text of the model appears along the left edge of the green box in quotes.

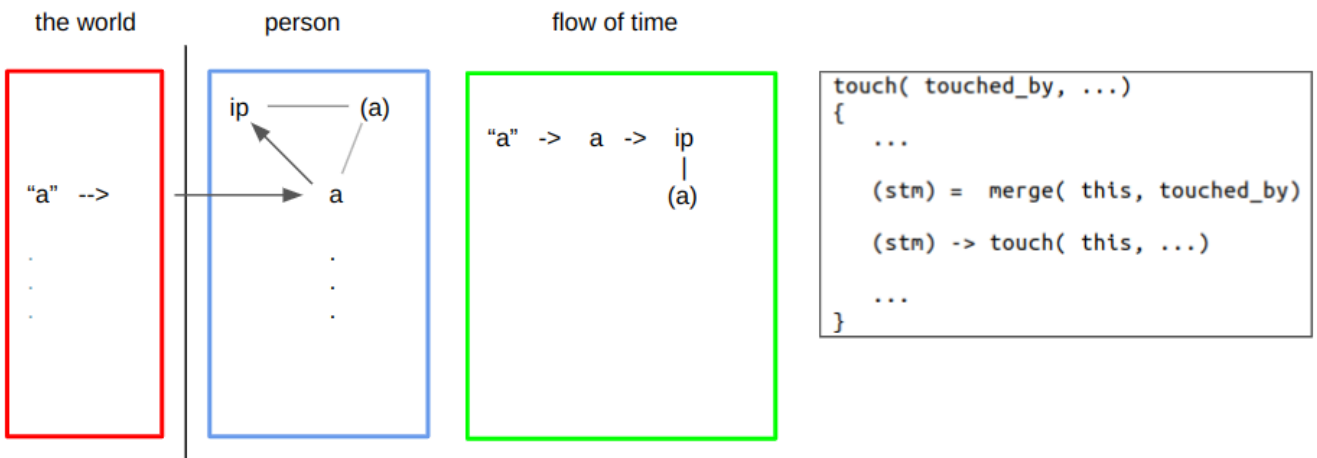


Figure 7 A “touched_by” variable is added to the touch() function call in order to do a merge to create the (a) node. This figure is analogous to figure (2) of (Chomsky et al. 2023), specifically the (stm) node (a) is the inscription created in the workspace.

In order to associate multiple symbols together in a sequence, we deduce the required dynamics of the model and introduce two new variables to the graph model. We introduce a direction parameter with two defined values (right, left) which will control functional flow in the touch() function³. Additionally we introduce a local variable to the Node{} class that stores a pointer to the last (stm) node made from the node. When a new (stm) node is created with the merge function, the pointer to the created (stm) node is set in the head and copy nodes for later use as shown below.

³ The right and left modes of the touch() function are shown to be analogous to the SMT External Merge and Internal Merge respectively below.

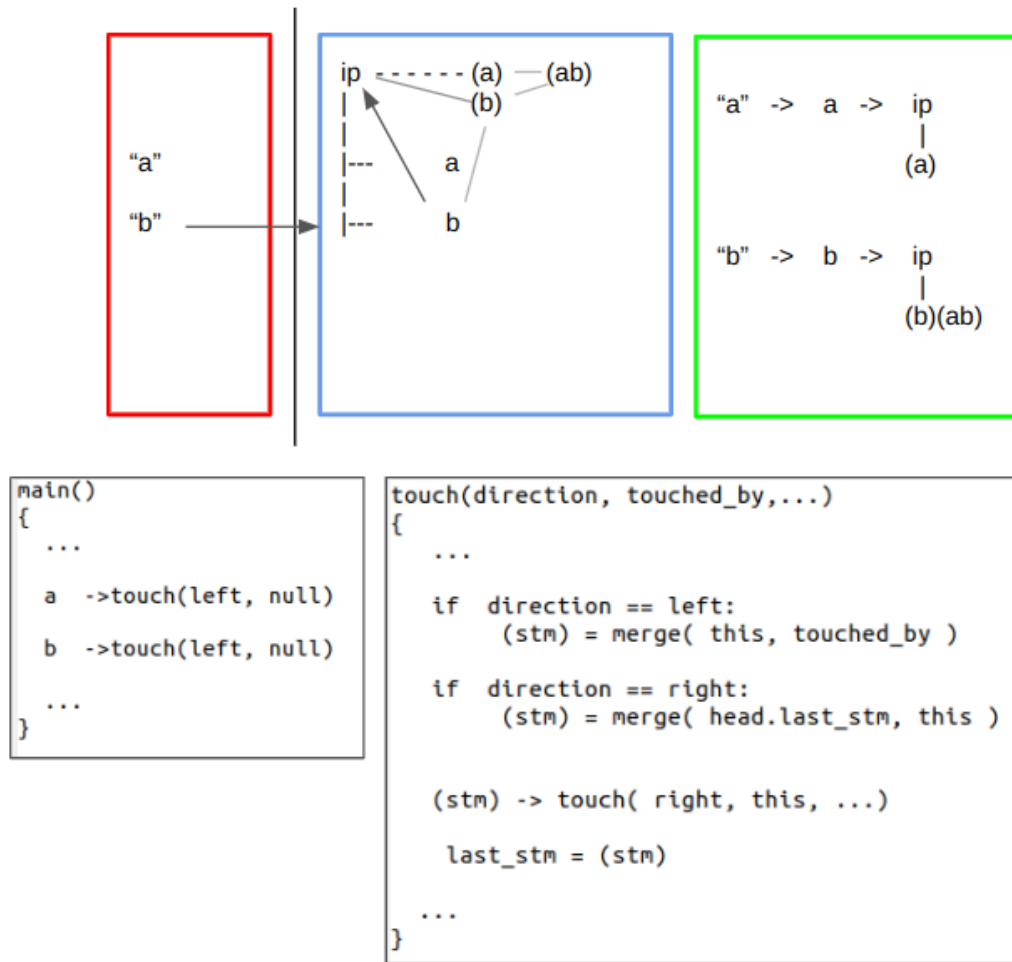


Figure 8 The (ab) node is created by a merge(h,c) function call using the stored "last_stm" variable (in the ip node).

The (stm) structure built on the two symbol input can be used to externalize (repeat back) the two symbols and to cause the (stm) structure to replicate itself. The `ab->touch(left, null)` call from `main()` in figure 9 will cause the touch function to touch its head, then copy. If the copy link == null, then the node is an interface node and the symbol is externalized (i.e. the symbol is printed in quotes along the left edge of the green box).

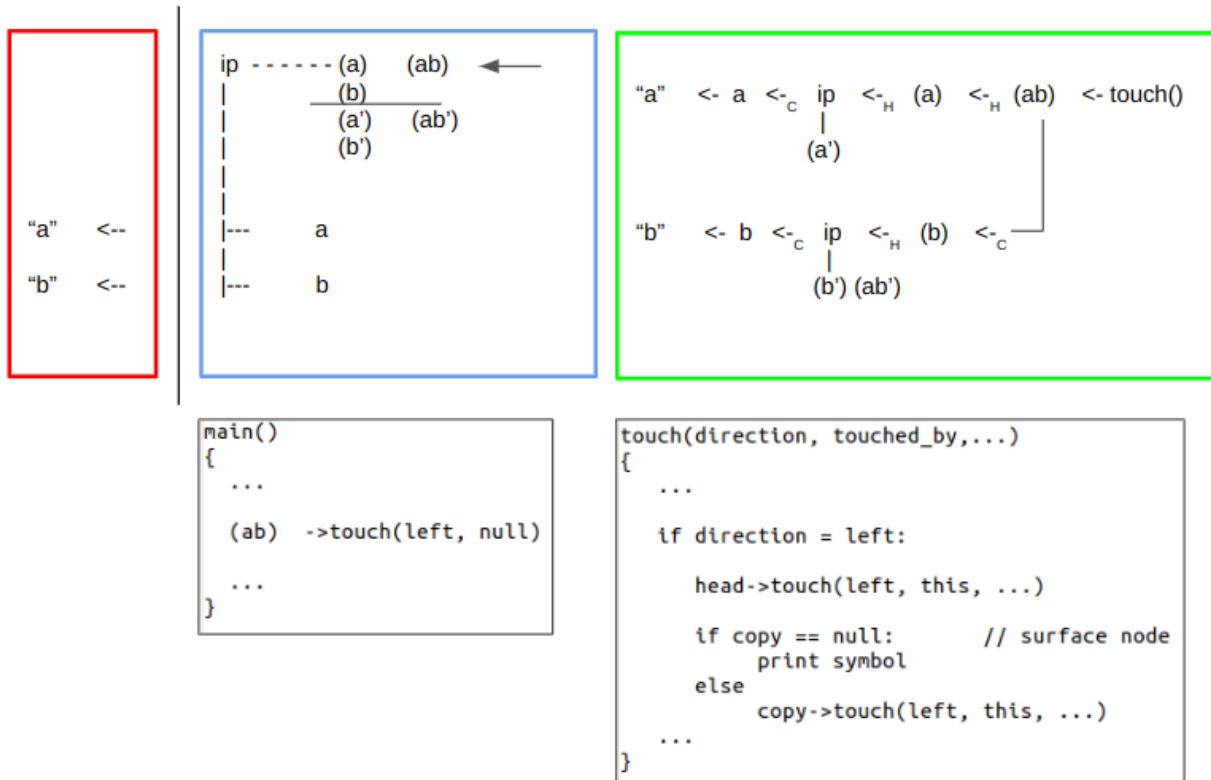


Figure 9 The (stm) structure is used to repeat back the two input symbols. Subscripts H and C are used in the flow diagram to indicate the head and copy touch calls from each (stm) node.

In (Del Signore 2024), wh-movement is implemented in the touch() function by switching the order of the head and copy touch calls in the (direction = left) code block above when an (stm) node's internal value parameter equals 0, which indicates a question. This allows for modifying the order of symbol output, ie: boys eat apples -> boys eat what? -> what boys eat? This movement logic is not included in the pseudocode above for brevity.

The (ab)->touch(left,...) function call from main in figure 9 can be accomplished from within the touch function with a single code change as given in figure 10. A "?" node is input in figure 10 that causes the stored (stm) sequence to be output. The last (stm) memory node made in the

right direction, $(?)_c$, will detect this state by the merge function returning $(stm) = \text{null}$, because $\text{head.last_stm} = \text{null}$, and can make a recursive: $\text{head.touch}(\text{left}, \dots)$ function call as shown.

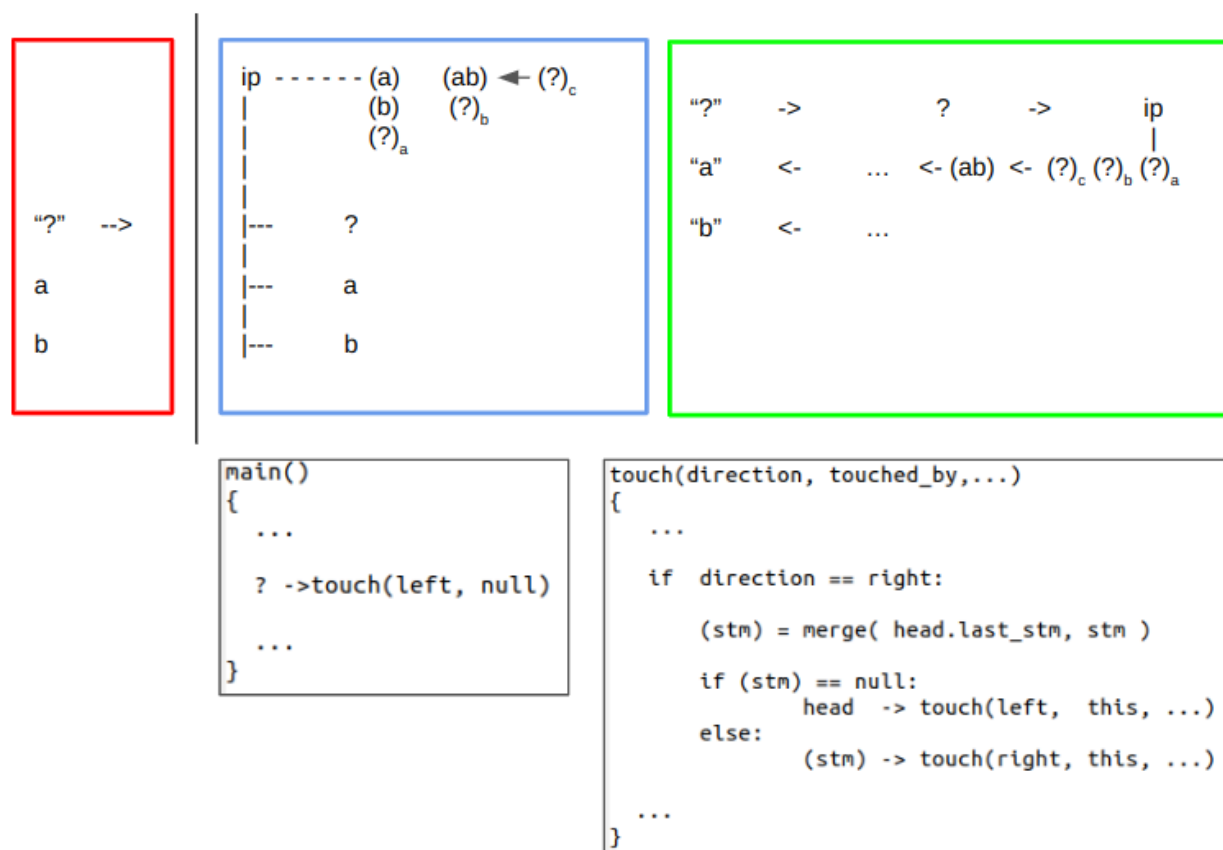


Figure 10 A ? node is input that causes the stored (stm) pattern to be output. The head link weight used by the $(?)_c$ node to touch the (ab) node can be inherited from the ? node. This would allow node specific behaviour based on the weight of the head link of a node. For instance the symbol nodes could be weighted such that they would not cause the touch(left) call to fire the (ab) node.

In (Del Signore 2024), the left direction touch call from the terminal (stm) node is also used to probe for closed loops in (stm) memory. This mechanism is described to detect illicit conjunction and direct object constructions such as: car and truck, car and *red, I quickly eat food, I eat *quickly food. The pseudocode for this functionality is likewise left out of the current paper for brevity.

In mammals, short term memories are accumulated throughout each day and then during a sleep cycle, the (stm) memories are replayed while dreaming which allows long term memories to form. In the graph model, long term memory nodes can be created from the merge() function as shown in figure 11. We define the rule that an [ltm] node is only created if one does not already exist. The touch function can connect the (stm) and [ltm] nodes at merge time as shown. The [ltm] node can then be conditionally fired at the end of the input cycle as shown.

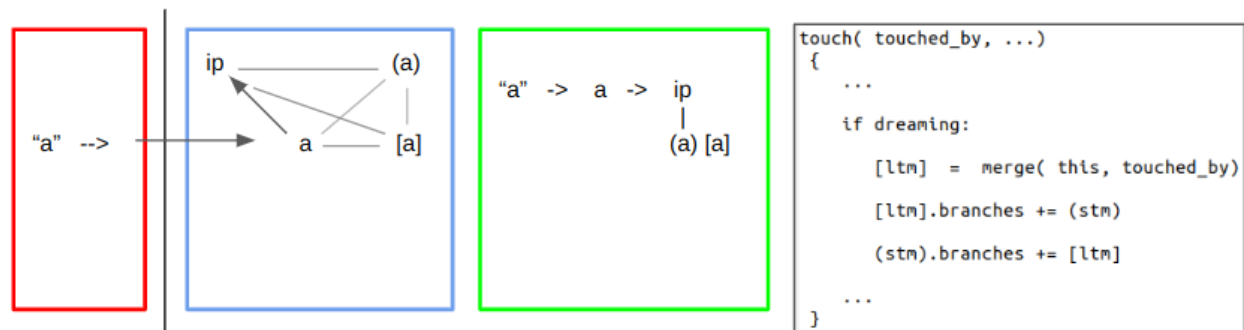


Figure 11: Long term memory node creation using a dreaming mode flag in the touch() function. The (stm) and [ltm] nodes can be connected at merge time. The [ltm] node can be fired at the end of the nominal input cycle.

An animation of this (stm) -> [ltm] process made with a graph model prototype is given in (Del Signore 2024a) using four random input symbols [hiker, rocks, duck, sneeze]. Figure 12 is taken from this animation and shows the model after the (stm) sequence has been replicated into [ltm] memory. The (stm) memory is then cleared and the [ltm] structure is used to recall the symbol sequence to (stm) memory.

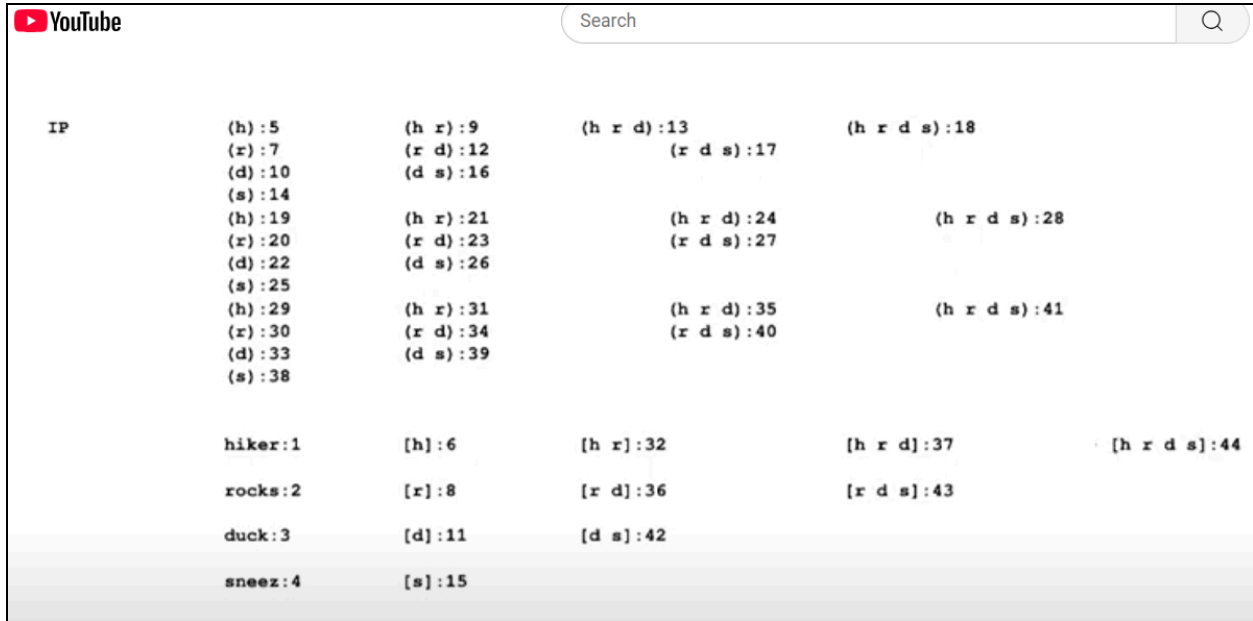


Figure 12: screenshot of a c++ animation of the graph model inputting four symbols into (stm) memory, and then replaying the (stm) sequence in order to form long term memories, denoted [ltm].

We next consider the input of illicit adjective sequences such as: “the big red dog” versus “the red *big dog”. In the graph model, the false indication (*) is posited to be generated by the merge function as shown in figure 13. This figure makes the assumption that the nodes [big, red, dog] have been sorted by an unspecified mechanism. If the internal variable that the nodes are sorted on is the head node weight, then the merge function can be made to compare the head weights of the head and copy nodes and to return a false indication if the weights do not follow some rule, such as $\text{head.weight} \leq \text{copy.weight}$.

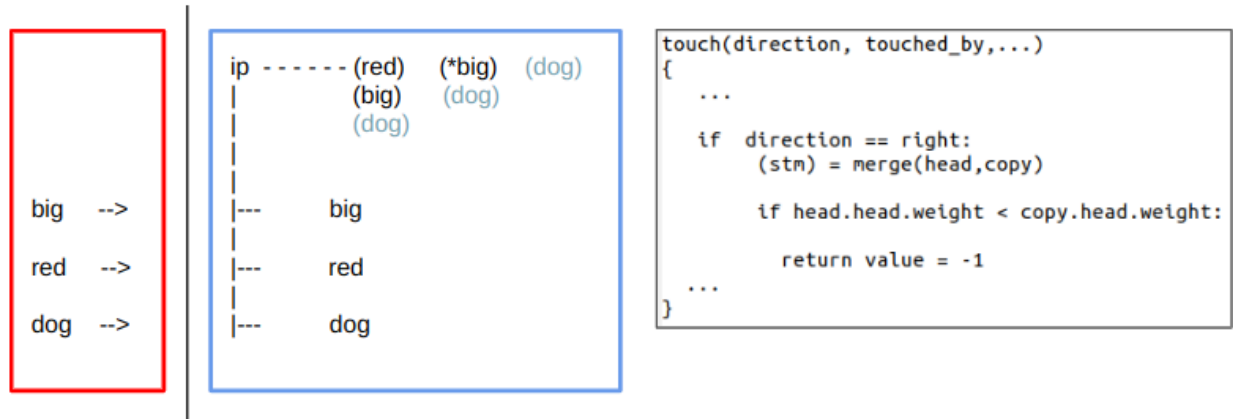


Figure 13 An illicit adjective sequence is posited to be detected at the return of the merge of the (*big) node, and to cause the return value of -1 from the touch function. This false signal could also be conveyed to main() though a global or stack variable.

All combinations of shapes, colors, and nouns can produce a false signal upon illicit input order, therefore it is highly improbable that a sorting mechanism could work across all individual interface nodes at the ip level. What is much more probable would be that the interface nodes merge with their respective hypernym nodes and the hypernym nodes are sorted, possibly via a run-time Hebbian mechanism⁴.

This graph representation is shown in Figure 14. All interface nodes first are formed as children of the ip node as before. The interface nodes can then do a merge with their respective hypernym nodes [size, color, noun]. For example big' = merge(size, big).

Upon input from main(), the touch function can now continue right to the big' node, or go up to the ip node. The supposition is that the weights and thresholds of the ip and big' nodes are such that the big' node would fire before the ip node in an incrementally increasing input scenario. Presumably if it would be evolutionarily advantageous for a node to continue firing to

⁴ in preparation.

the right into its hypernym branch, then some Hebbian mechanism would set the weights accordingly.

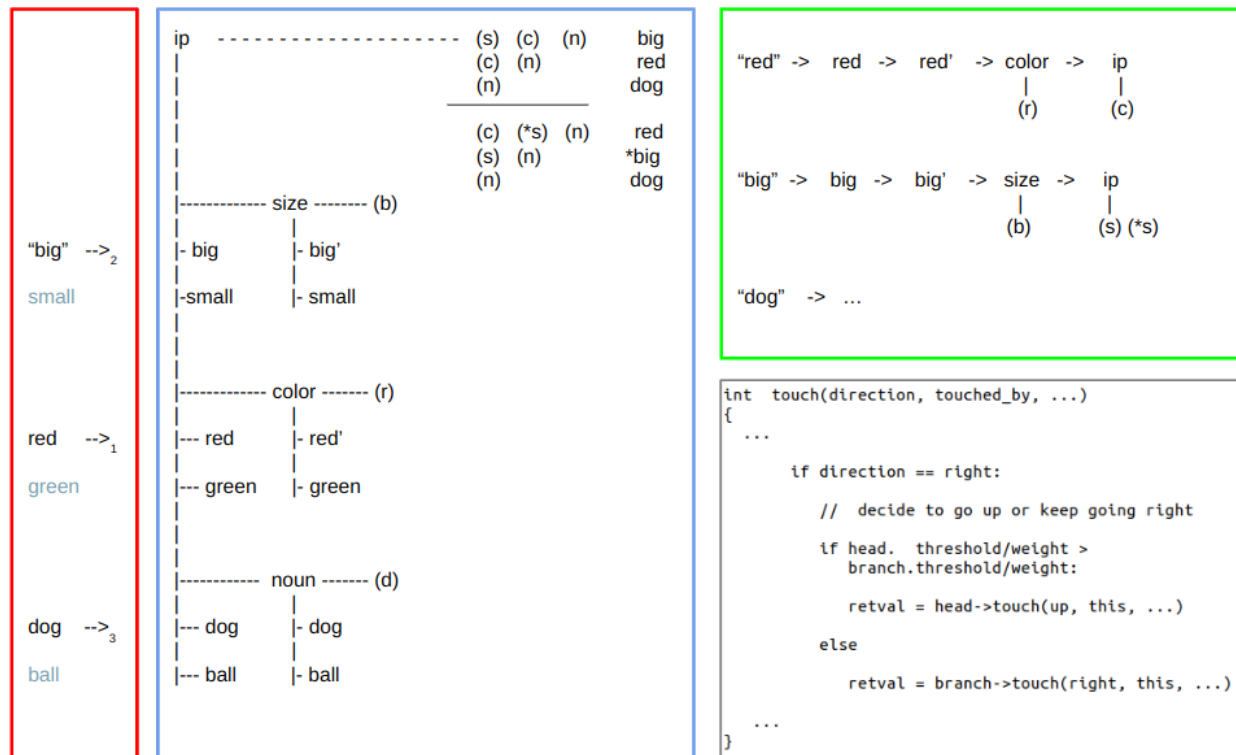


Figure 14 The model assumes that some internal parameter is changing in a consistent manner across the hypernym level nodes [size, color, noun] and this parameter is inherited by the (stm) nodes and then used by the merge function to detect an illicit ordering.

In the above example the (*) stm node is created in the ip node's child (stm) nodes. Figure 15 describes a longer input sequence in which the (*) nodes are created at the lower level graph branches. In this example, each input symbol causes the order of node firing to go right until the terminal node is reached, or some other condition, and then to fire left, to the ip node. The child nodes at each level of the graph are not shown for brevity, except for the steel node.

We next examine the process of counting in the graph model. fMRI experiments studying counting in the neocortex are well developed. In (Vogel 2017), multimodal activation of a single area in the neocortex, the IPS area, in response to numeric input is observed. Input can be verbal or sight, with written characters or with some number of dots. In the graph model, input is analogous to a written stream of symbols. Each symbol is input from main{} via symbol->touch(right,...) function calls⁵.

Figure 16 shows a graph containing the symbols 0 through 9. Merges are done in main() between each symbol to produce the primed copy of each node. This network then can be made to produce the output of counting from 0 to 9 repeatedly, forever. We can note that (stm) nodes are created during the counting process, but never re-used.

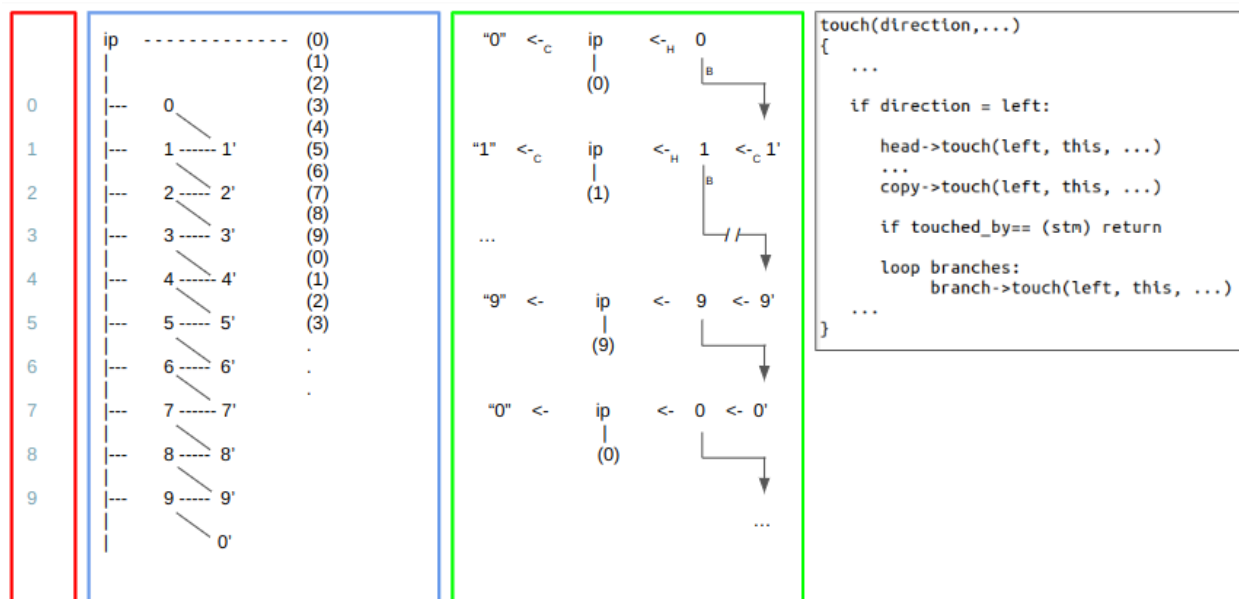


Figure 16 The network functions as an infinite loop that outputs 0 - 9 repeatedly.

⁵ Visual input of numeric information, such as some number of dots on a line, is discussed in the next steps section.

In the pseudocode in figure 16, a loop is added over the node's branches and a touch() call is made to each branch. The loop is not done if the touched_by node is an (stm) node; this preserves the existing touch(left) functionality. This code is added to the (direction = left) code block of the touch function, although the same functionality could be done in different ways, such as by defining a direction = any code block in the touch() function.

We next add a single node with a modified threshold value to this network and it will change the network behaviour from counting from 0 to 9 forever to counting from 0 to infinity. The single node is denoted in bold as "**1**" in figure 17 and it has the property that when it is inserted into the digit stream, it will re-copy itself into (stm) memory each time it is externalized, leading the to **1** node being inserted between each digit of the original sequence. The output sequence becomes: 0 1 2 3 4 5 6 7 8 9 **1** 0 **1** 1 **1** 2 **1** 3 **1** 4

A reduced node threshold can be used to cause the desired behaviour. In Figure 17, we can see that the node (0)_c touches(left) the (**1**)_a node. We make the assumption that the threshold of the bolded (**stm**) nodes is such that they will fire when touched(left), leading to the externalization being repeated and a new bolded (**stm**) node inserted into the ip (stm) workspace.

The reduced threshold can be inherited from the **1** node via the (**1**)_x node if we add a line of pseudocode to the touch function that copies the threshold at merge time from the last_stm node of the copy node used in the merge. The modified section of pseudocode is given in figure 17. If we make the assumption that all of the surface and primed nodes have the same initial threshold, then this change to the pseudocode will not affect existing behavior.

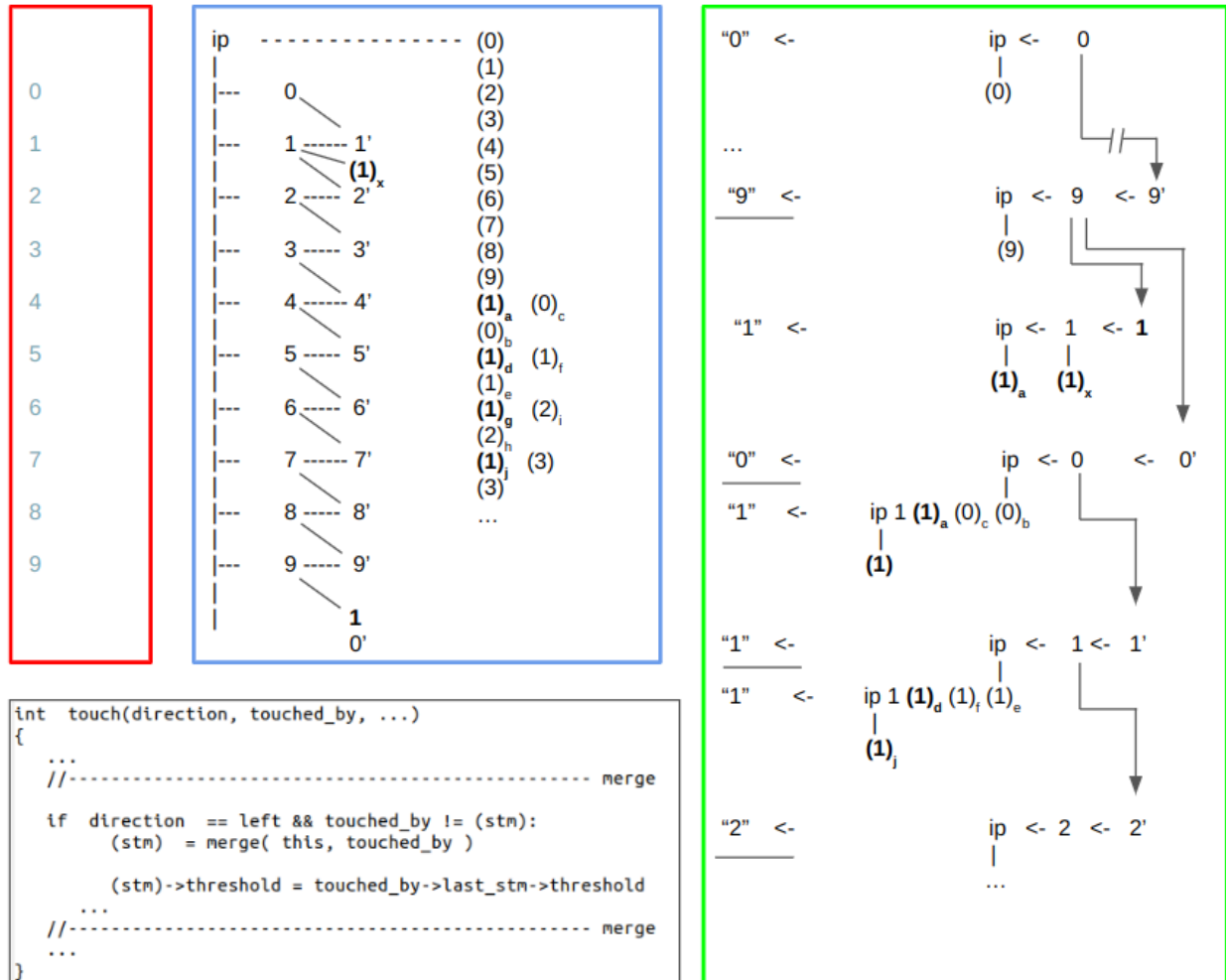


Figure 17: 9 to 10 transition: The 9 node externalizes and then touches and fires its branch nodes, causing “1 0” to be output and inserting the **(1)_a** node into (stm). The (0)_c node fires the **(1)_a** node, which causes it to touch and fire the 1 node, which causes a new **(1)_d** node to be inserted into (stm) memory. Each time a 0-9 node fires, it will cause the last **(1)** node to refire and output a “1” and reinsert a new **(1)** node into the ip node’s (stm) list.

When the sequence gets to 19, a change in node firing order occurs to advance the bolded digit to a **(2)**. This is shown in figure 18. When the 9 node fires and externalizes, it fires the node sequence (9)_e (9)_f (1)_d 1. At the 1 node, we can deduce that the node should not externalize via the head and copy touch calls, as if the nodes were touched but did not fire, and instead the 1 node should touch and fire the branch node 2'. This leads to a “2” being externalized and a **(2)** node inserted into (stm) memory as shown. This implies that the bold

property of the **(1)_d** node is passed to the **(2)_g** node through the inheritance mechanism at the merge.

This functionality at the 1 node could be accomplished if the weights used to touch the head and copy node are reduced such that the head and copy nodes do not fire. But this would break the 0-9 infinite loop functionality. We can deduce from figure 18 that the rule should be that when the **(9)_f** node touches the **(1)_d** node with direction = left, that some information is passed in the touch() call, and this information is then passed to the 1 node, causing the head and copy not to fire.

One way this could be accomplished could be by inheriting the head and copy node weights from the touched_by node in the direction=left block of the pseudo code in figure 16. These values would be originally inherited from the 9 surface node in figure 16, and they would be unique to the 9 node so as to produce the desired effect of incrementing the bolded node. The head and copy touch function calls in the pseudocode in figure 16 would be modified from:

(head/copy)->touch(..., weight = (head/copy).weight)

to:

(head/copy)->touch(..., weight = touched_by->(head/copy).weight)

In Figure 19, we see that when control returns to the 9 node after the head and copy touch calls, that the **1** node does not fire and instead the 0' node fires and never returns process control back to the 9 node. One way that the **1** node could be made not fire in this scenario is to set the node.value parameter with a flag when the bolded (**stm**) digit advances, and to use this flag to reverse the order of the loop over the branches in the left branch of the touch function. This functionality is implemented with two lines of pseudocode in figure 20, one that sets value =-1,

and one line to use the value=1 condition. The value =-1 code runs at node 1 in the sequence (9)_e (9)_f **(1)_d** 1 in figure 18.

In (Del Signore 2024), the same logic applied to the head and copy touch function calls to implement movement, specifically the order of the function calls is reversed in the direction=left leg if (stm).value = 0.

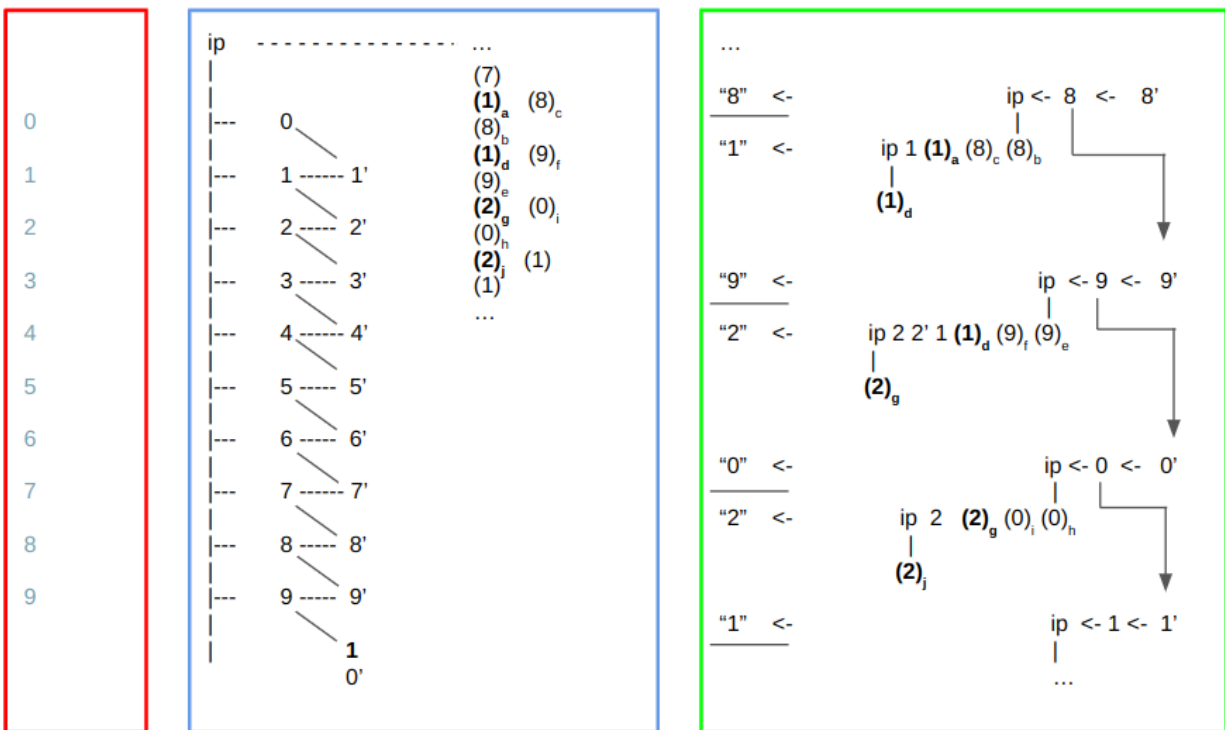


Figure 18: 19 to 20 transition. The rule is when a (9) touches a bolded (stm) node with direction = left, the bolded (stm) node will not fire its head and copy nodes and instead will fire its branch nodes.

The counting sequence then proceeds to 99 and makes the transition to 100 as shown in figure 19. No further parameters or code changes are needed. In the (9) (9) **(9)** 9 sequence in figure 19, the 9 node will not externalize its head and copy links. The 9 node then fires its branches in original order producing the output: 1 0. The 9 node then returns a value = -1 based on its head and copy not firing. Control returns back to the 9 node through the ip node

and the 9 node will reverse the order of the loop over the branches, causing the 0' node to fire and take program flow control and to never return.

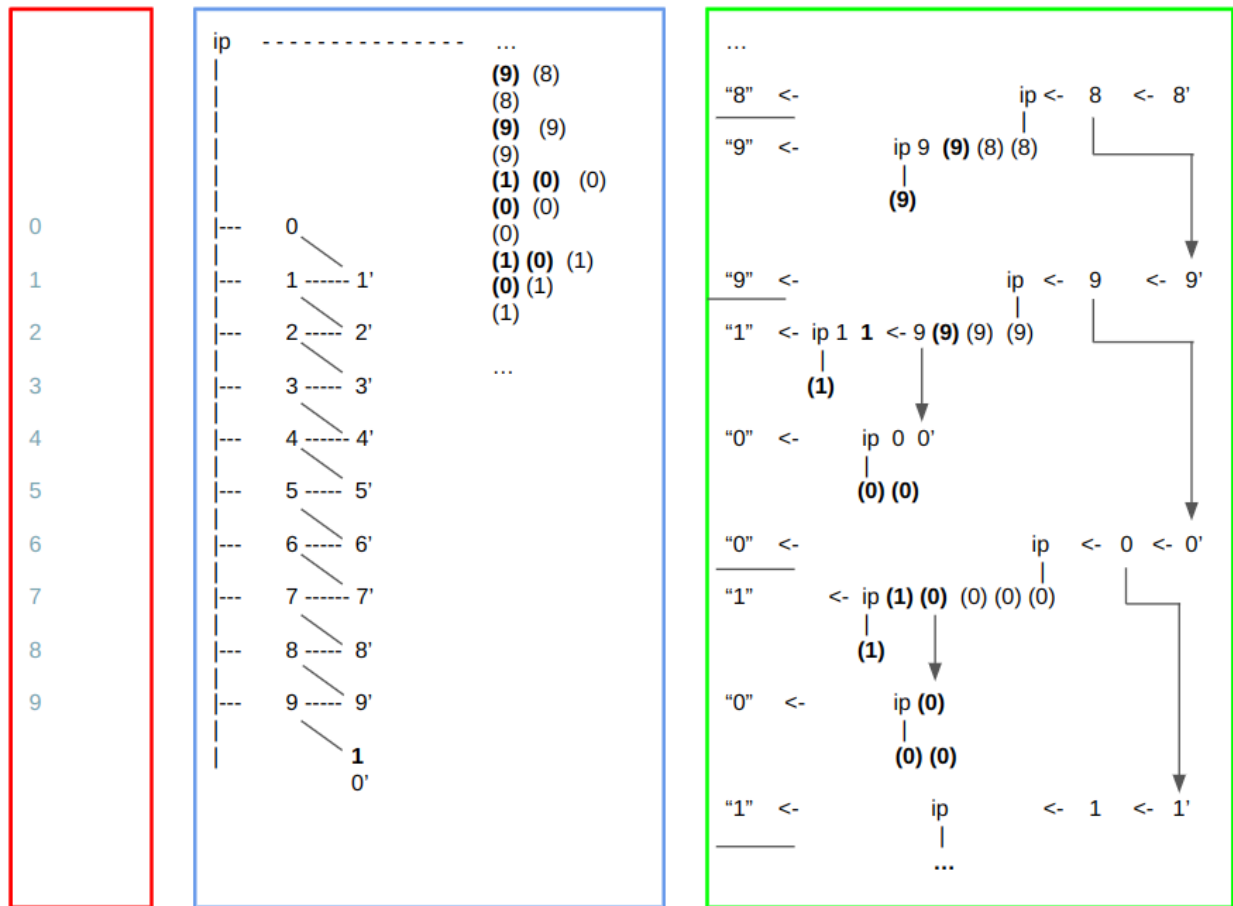


Figure 19 99 to 100 transition.

The pseudocode for the touch() function is shown in figure 20 with the Node{} class given in figure 5. The development of this code occurs incrementally as graph dynamics are developed to replicate the Human language examples. This process could be considered a form of intelligent design by incremental reverse engineering.


```

int touch(direction, touched_by, ...)
{
    if node fires:

        //----- merge
        if direction == left && touched_by != (stm):
            (stm) = merge( this, touched_by )

            (stm)->threshold = touched_by->last_stm->threshold

        if direction == right && touched_by == (stm):
            (stm) = merge( head.last_stm, this )

        (stm) -> touch( right, this, ...)

        last_stm = (stm)

    // ----- left
    if left:

        retval = head->touch(left, this, touched_by->head.weight,...)

        if copy == null:      // this = interface node
            print symbol
        else
            retval = copy->touch(left, this, touched_by->copy.weight,...)

        if (head & copy both do not fire) value=-1

        if touched_by== (stm) return

        loop branches: (reverse loop if value=-1)
            branch->touch(left, this, ...)

    // ----- right
    if right:

        // decide to go left/up or keep going right

        if head. threshold/weight >      // 1st to fire in
            branch.threshold/weight:      // incremental
                                           // input scenario

            retval = head->touch(up, this, ...)

        else

            retval = branch->touch(right, this, ...)

    // ----- return

    if (value = -1) touched_by->value = value

    return retval
}

```

Figure 20 Pseudocode of the touch function. Low level functionality is excluded for simplicity, such as null pointer checks; additionally the touch() calls will be skipped if the node being touched is the touched_by node.

The pseudocode in figure 21 shows a progression of main1-6() routines. The Node class and merge function are initially developed for use in main1(). As more language experiments are implemented the Node class is evolved to run the examples in each main#() routine.

Regression testing can be done by changing to comment lines in main(). In nature, we would generally expect the existing functionality of the Node class to be preserved as it evolved.

```
Node{...}
merge(h,c)
main1(){...}
main2(){...}
main3(){...}
main4(){...}
main5(){...}
main6(){...}
main(){
  //main1();    // hiker rock duck sneeze   (stm) memory
  //main2();    // a-n graph, (stm) + [ltm] memory
  //main3();    // movment, boys eat what?,   what boys eat?
  //main4();    // a-n graph + conjunction
  //main5();    // movie mode, hiker rocks duck sneeze
  main6();     // adjective sequences
}
```

Figure 21 Evolutionary programming paradigm.

The code changes at each step can be very specific and complex. These are reverse engineered from the language examples under study. The hypothesis is that in nature, all possible mutations of the system can occur and cause small changes, and then if the small change is beneficial, it persists. So the evolution of language could have occurred as a sequence of small changes in the neocortex neural firing dynamics.

Comparison with the SMT Program

The fundamental proposition of the SMT is that all of human language can be generated via a single recursive Merge(head, copy) function. The Graph Model, which was developed largely by following the problems raised by the SMT program, posits that it is all of mammalian complex behaviour that can be represented in the same manner and that language is a recent extension of the capabilities of the mammalian neocortex.

The graph model encloses a merge(h,c) function within a touch() function which controls the application of merge within the lexicon and workspaces. In the SMT model, these control functions are described as happening from within the merge() function. In practice there is no difference. In the graph model code, the merge function could be directly copied into the touch function in the two places it is called and then the touch() function could be renamed Merge(). The graph model architecture of enclosing the merge function within the touch function, touch{merge{ } }, has evolved out of normal computational structural considerations.

In the graph model a weight is given to each link, in each direction, and a firing threshold and a floating point value are given to each node, as shown in Figure 22. The values of these parameters can control the output ordering during externalization and graph flow in the lexicon and workspace during input. These parameters are also modified at input time with a Hebbian mechanism under certain conditions. This is analogous to Short Term Plasticity observed in neocortex neurons⁶.

⁶ If we assume the neocortex is exposed to a sequence of symbols and that not all of the information is important, then a function of Short Term Plasticity can be argued to be to bind together the important symbols that should be retained, immediately at input time.

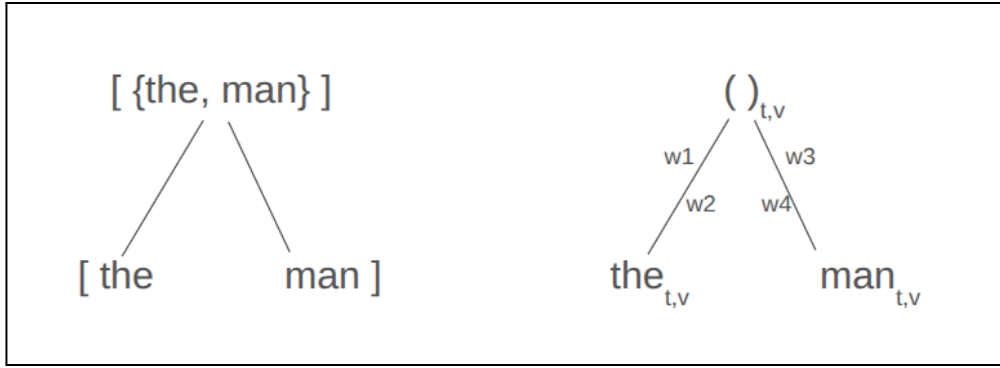


Fig 22 The SMT merge operation compared with the graph model merge. The nodes in the graph model all have a threshold and value parameter. There are four weight variables introduced for the two bi-directional links. These weights are initialized with inherited values and can then be modified with a Hebbian process.

In the SMT model, recent work with non-planar trees leaves the link parameters unassigned until final externalization (Marcolli et al.). This allows for displacement to be solved in the similar manner as in the graph model (Seely 2023), namely by reversing the order of the `head.touch(left)` and `copy.touch(left)` calls in the touch function.

Workspaces and a search function are introduced into the SMT model in (Chomsky 2019). The search function searches the workspace for a pair of inscriptions to merge. First principles considerations of efficiency constraints on the search function lead to the definitions of Internal Merge and External Merge:

With EM the two Merge targets P, Q are separate members of the WS. With IM, Q is contained within P .

These definitions suggest that the left and right modes of the `touch()` function correspond to IM and EM respectively. These processes are shown in Figure 23. Upon input at the interface, an interface node is touched and fired in the right direction. The touch function then

determines whether to keep going right into a hypernym branch or to go left, up the graph to the ip node. This decision is based on the local link weights and thresholds in each direction.

As drawn in Figure 23, the touch function continues right until it reaches a terminal node and then switches to the left direction and traverses left and up the graph towards the ip node. In this left mode, an inscription into each local workspace is made by an Internal Merge between each head and child lexicon node. The four nodes along the left edge of each workspace in Figure 23 are all made with Internal Merges. These are the initial inscriptions made in the workspace, always on the start of a new row.

When a new (stm) node is added to the workspace, it is touched and fired⁷. At the new (stm) node, if the touch() function finds a valid last_stm pointer in its head node, it does an External Merge as: $(stm)_{new} = \text{merge}(\text{head} \rightarrow \text{last_stm}, \text{this})$ ⁸. Following this merge, the new (stm) node is touched and fired and the process repeats, adding a new (stm) node, via EM, to the end of each existing row of (stm) nodes in the workspace⁹. In this manner the workspace is searched by the touch function. The accessible inscriptions are the (stm) nodes at the right side of every row of (stm) nodes in each workspace.

⁷ A maximal efficiency argument can be made for touching the new workspace node as soon as possible after it is created, namely that it takes energy to do a merge and form a new node and any value that the new node could add to the system happens because the node can fire and merge with other workspace nodes, therefore the new node should be exploited immediately.

⁸ The last_stm pointer is then updated to point to the new (stm) node after the merge() operation.

⁹ The ()_{r-c} node in figure 23 is made with an EM.

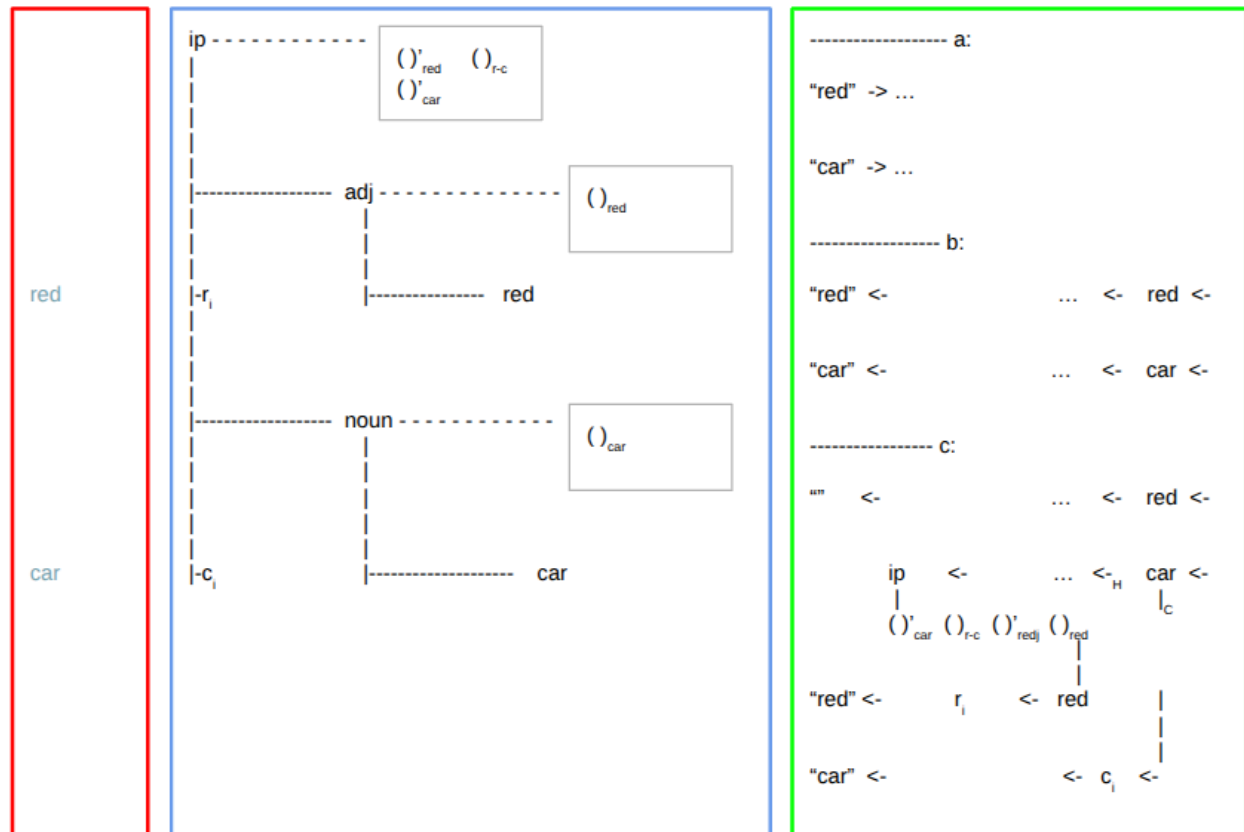


Figure 23: The noun phrase “red car”. In example a:, “red car” is input from the interface. In example b:, the lexicon nodes red and car are touched(left) and cause “red car” to be externalized. Both examples cause the exact same workspace configuration of (stm) nodes to form as shown. Example c: is the same as example b:, except that “red” is not externalized at the interface as indicated by the double quotes¹⁰. When the car node is touched and fired, the touch() function finds the ()'_{red} node in the ip workspace and causes “red” to be externalized.

Figure 24 shows the example noun phrase: “the two red cars”. The symbols two, red, and cars are first premeditated to create the (stm) nodes in the workspaces as shown. The car node is then touched in the left direction. The touch() function then traverses the lexicon and workspace towards the ip node and finds the (stm) memory nodes in each workspace and can externalize on each one.

¹⁰ By the touch() function suppressing the externalization at the interface, which could be based on some combination of local and global variables. This mechanism can also produce ellipsis phenomena such as “you give me the thing” versus “give me the thing”.

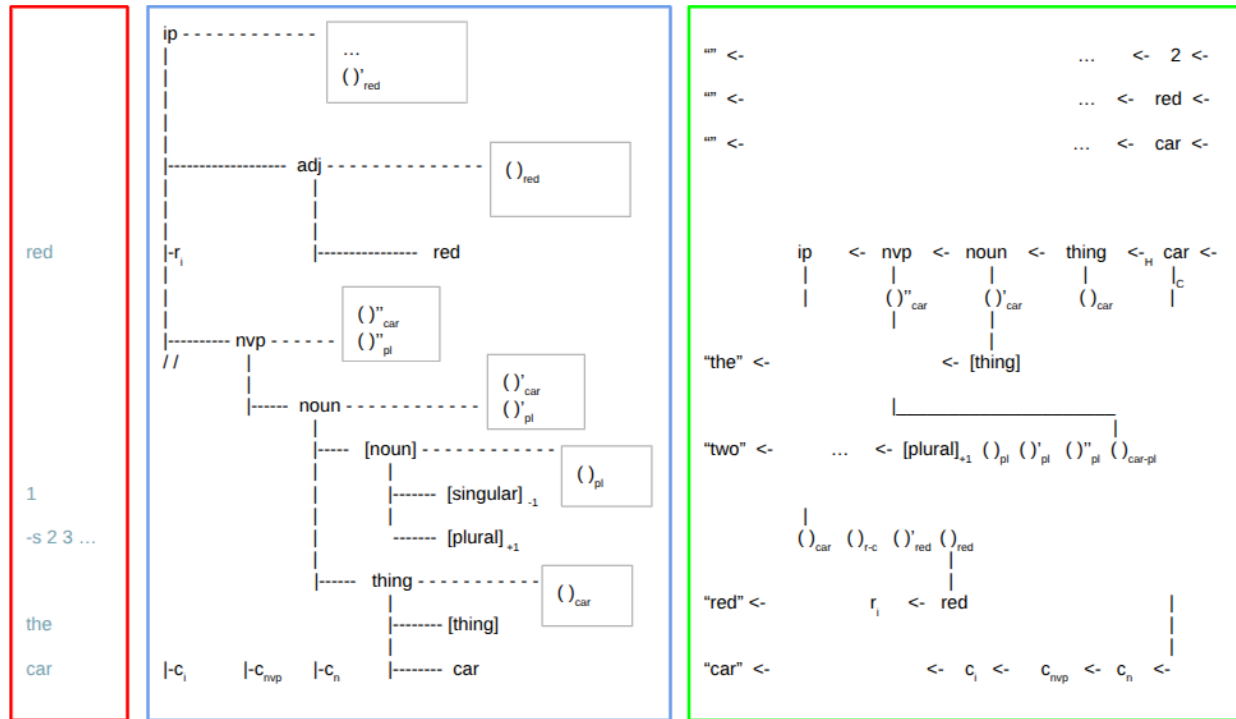


Figure 24: Example of premeditating and externalizing a noun phrase. The nodes 2, red, and car are first internalized by touch(left) function calls with suppressed output. This is assumed to leave the (stm) memory in a state that it can still be externalized; following which the car node is touched and the touch() function traverses the graph towards the ip node to generate the text “the two red cars”¹¹.

Summary and next steps: The graph model can account for all low level language phenomena so far examined. These are summarized in Table 2. The behavioral sequences so far described are simple language interactions based on information stored in short term memory. Long term memory [ltm] nodes are introduced in figure 11. These can be conditionally fired, presumably after an initial input cycle. A minimum behavioural interaction with [ltm] would be: input -> ... -> [ltm] recall -> ... -> output.

The terminal nodes of the graph must be extended to add further functionality. New terminal nodes can be added manually via merge(h,c) operations using existing parent nodes. The extensions can be made by deduction. In the case of “the third yellow house”¹², there is a single change from figure 24, in which the phrase would be “the three yellow houses”. The

¹¹ The generation of the -s plural word ending is not shown and is left as an exercise for the reader.

¹² (Chomsky et al. 2023) example (11)

ordinal series first, second, third.... could be made by merges between the number series in figure 24 and the [singular] node. This would allow detection of the illicit phrase: the third yellow houses*.

Hypernym associations such as “car is a vehicle” can form as [ltm] nodes in the nvp branch of the graph shown in figure 15. Possession verbs appear to form associations in the noun branch, and possibly lower, in figure 15, such as: car of mine -> my car.

Functionality can be added to the verbs and prepositions by merging the hypernym verb and preposition nodes with other hypernym nodes in the graph. In the case of the verb get, a new child node made as get' = merge(get, noun), which can be touched and fired following a successful input of get-verb phrase and would cause a possessive association¹³¹⁴ to form in the noun workspace between the subject and object.

The mammalian visual area v1 is a two dimensional neural network mapped directly to the field of view of the retina and the v1 parcel on the neocortex (Kruger,et al. 2013) Such a structure could evolve in the graph model from merges between the “space” and “sense” nodes in figure 15. The v1, v2, v3, v4 structure of the neocortex visual area is also amenable to construction via merge operations¹⁵.

¹³ by an external merge in the noun workspace, similar to (Chomsky et al. 2023) example (40)

¹⁴ This EM is equivalent to a possessive theta-role merge, always between two nouns in the noun workspace for “noun1 get noun2” phrases.

¹⁵ in preparation; the counting of dots and connection to a multimodal area as in (Vogel 2017) is an early design goal.

Table 2:

Language Phenomena	Example	Graph Model Implementation:
sequence input and storage	man eat thing	(m) () () (e) () (t)
sequence recall	man eat thing	(m) () () <- touch() (e) () (t)
questions	man eat what?	(m) () () ₀ <- touch(0) (e) () ₀ (t) ₀
wh-movement	man eat thing what man eat? man eat meat what man eat?	if value=0 touch copy node 1st: () _{man} () () ₀ <- touch() () _{eat} () ₀ () _{thing(0)} thing(0) externalizes to “what” meat(0) externalizes to thing(0)
adverb periodicity	<div style="text-align: center;">a n a n a</div> <div style="text-align: center;">-----</div> <div style="display: flex; justify-content: space-between;"> <div>quickly</div> <div>I</div> <div>eat</div> </div> <div style="display: flex; justify-content: space-between;"> <div></div> <div>I</div> <div>quickly eat</div> </div> <div style="display: flex; justify-content: space-between;"> <div></div> <div>I</div> <div>eat quickly</div> </div>	ip -----adj ----- quickly ----- nvp ----- I ----- eat
conjunction	Wallace and Gromit ¹⁶	Hebbian binding of closed loop in (stm) memory ip ----- noun - - - (w) (wg) (g)

¹⁶ (Phillips, 2013)

illicit conjunction	Wallace and quickly*	non closed (stm) loop: ip ----- adj - - - - (q) ----- noun - - - - (w)
movement of conjunctions	man eat meat and fish? meat and fish man eat?	(m) () () () () ₀ <- touch() (e) () () () ₀ (m) ₀ () ₀ () ₀ (+) () ₀ (f) ₀
direct object	man eat meat	uses the same Hebbian closed (stm) loop mechanism as the conjunction
illicit direct object	man eat *quickly meat	same non closed (stm) loop mechanism as the illicit conjunction
compare function	here HERE	merge function detects change in value of head and copy nodes: (h) (+/-) (H)
past and future tense	man will talk ed*	The [ltm] node of verb bifurcates on value to represent tense: ----nvp -- verb ----[verb] - - - - (+) (*) “will” --- [+] (-) “ed” --- [-]

irregular past tense	<p>ate eat-ed</p> <p>$[-]_e = \text{merge}(\text{eat}, [-])$</p> <p>on input, “ate” produces the same (stm) configuration as verb + “ed”</p>	<p>The past node $[-]$ merges with eat to produce the $[-]_e$ node:</p> <pre> // -- verb ----[] “ed” ----- [-] eat ---- eat ate ----- [-]_e </pre>
progressive and perfected	<p>Wallace and Gromit are talking</p> <p>Wallace and Gromit have talk ing*</p>	<p>The [itm] node of the nvp node bifurcates on value to represent the progressive and perfected tenses:</p> <pre> -----nvp ----[nvp] - - - - - (-) (*) (+) “are”,ing --- [+] “have” --- [-] </pre>
singular and plural counting	the one thing s*	<p>The [itm] node of the noun node bifurcates on value to represent singular and plural:</p> <pre> ----- noun ----[] - - - - - (-) (*) (+) 1, singular --- [-] 2, -s, plural --- [+] 3 [+] 4 [+] 5 [+] ... </pre>
adjective ordering	<p>the big red dog</p> <p>the red big* dog</p>	<p>Hebbian mechanism sorts hypernym node weights per customary usage. Merge function detects out of sequence symbols and returns false.</p>

ellipsis	you give me the thing please give me the thing please	The touch function detects the interface by NULL copy pointer and can then suppress externalization based on local and global variables.
Antecedent-Contained deletion	woman got the thing man got woman got the thing man did	creates the same possessive association in noun branch: woman's thing
Across-the-Board movement out of a conjunction	man eat meat and drink water what man eat and drink?	$()_{\text{man}} \quad () \quad () \quad () \quad () \quad ()_0 \leftarrow$ $()_{\text{eat}} \quad () \quad () \quad () \quad ()_0$ $()_{\text{meat}(0)} \quad ()_0 \quad ()_0 \quad ()_0$ $()_{\text{and}(0)} \quad () \quad ()_0$ $()_{\text{drink}} \quad ()_0$ $()_{\text{water}(0)}$ man drink what and eat what? what "and what" man eat and drink? what man eat and drink? -the "and what" is suppressed by ellipsis
Parasitic gaps	man get bike to ride it what bike man get to ride it? what bike man get to ride?	ellipsis of a pronoun $()_{\text{bike}(0)}$ is externalized as "what bike" using the same graph flow as "the bike", where $()_{\text{the}(0)}$ externalizes as "what"

List of Abbreviations

(stm)	short term memory
[ltm]	long term memory
main{}	The main function in c++
Node{}	The Node class
touch()	The touch member function of the Node class.

Authors' Contribution: The author confirms sole responsibility for the conception, design, literature review, analysis, interpretation, manuscript drafting, critical revisions, and final approval of the article.

Availability of Data and Materials: github links are provided in the references to the code used in the (stm) memory simulation and to the c++ code that implements the (stm) to [ltm] video animation.

Funding: No external funding was received for this research.

Figures Originality: The Author confirms the figures are all original and contain no copyrighted material.

Acknowledgement: The author thanks Chris Hyuak for his review and input.

References: (alphabetical)

Buzsáki G. (2011). Neural syntax: cell assemblies, synapsembles, and readers. *Neuron*. 2010 Nov 4;68(3):362-85. doi: 10.1016/j.neuron.2010.09.023. PMID: 21040841; PMCID: PMC3005627.

Chomsky, N. (2019). Some Puzzling Foundational Issues: The Reading Program. *Catalan Journal of Linguistics*, 263–285. <https://doi.org/10.5565/rev/catjl.287>

Chomsky N, Seely TD, Berwick RC, et al. (2023). *Merge and the Strong Minimalist Thesis*. Cambridge University Press; 2023.

Del Signore, K. (2024). How Language Could Have Evolved. In: Samsonovich, A.V., Liu, T. (eds) *Biologically Inspired Cognitive Architectures 2023*. BICA 2023. Studies in Computational Intelligence, vol 1130. Springer, Cham. https://doi.org/10.1007/978-3-031-50381-8_27

Del Signore (2024a). Animation of short and long term memory made with c++ prototype. https://www.youtube.com/watch?v=s6fIQ1q5i_g

Del Signore (2025). code used to generate figure 3: <https://github.com/kwd2/graph1/blob/main/stm.C>

Fitch, W.T. (2017). Empirical approaches to the study of language evolution. In *Psychon Bull Rev*. 2017 Feb;24(1):3-33. (2017). doi: 10.3758/s13423-017-1236-5 (2013): 263-288.

Hawkins, J., Blakeslee S. (2005). On Intelligence. Published by *Times Books*.

Huth A.G., et al. (2013). A Continuous Semantic Space Describes the Representation of Thousands of Object and Action Categories across the Human Brain. In *Neuron*. 2012 Dec 20; In 76(6): 1210–1224. doi:10.1016/j.neuron.2012.10.014. See figure 2, PPA analysis.

Huyck, Christian R., and Peter J. Passmore. (2013). A review of cell assemblies. *Biological cybernetics*. 107(3). DOI:10.1007/s00422-013-0555-5.

Ji, Y., Gamez, D., Huyck, C. (2024). Associative Memory with Biologically-Inspired Cell Assemblies. In: Samsonovich, A.V., Liu, T. (eds) *Biologically Inspired Cognitive Architectures 2023. BICA 2023. Studies in Computational Intelligence*, vol 1130. Springer, Cham. https://doi.org/10.1007/978-3-031-50381-8_43

Kruger N., et al. (2013). "Deep Hierarchies in the Primate Visual Cortex: What Can We Learn for Computer Vision?". In *IEEE Transactions on Pattern Analysis and Machine Intelligence*, vol. 35, no. 8, pp. 1847-1871, Aug. 2013. doi: 10.1109/TPAMI.2012.272.

Marcolli Matilde, Noam Chomsky and Robert C. Berwick. (2025). Mathematical Structure of Syntactic Merge, August 5, 2025, *The MIT Press*. ISBN: 9780262552523

Pengyu, Liu, et al, (2025). A Survey on fMRI-based Brain Decoding for Reconstructing Multimodal Stimuli, <https://arxiv.org/html/2503.15978v1>

Phillips, Collin, 2013, Derivational Order in Syntax: Evidence and Architectural Consequences. In *Studies in Linguistics* 6: 11-47.

Seely, T. Daniel, 2023, [Lecture 1]Working Toward the Strong Interpretation of SMT, https://www.youtube.com/watch?v=vD1Vqv7cmq4&ab_channel=DMCKeioUniversity, 54:30

Vogel, Stephan E. (2017) et al. "The left intraparietal sulcus adapts to symbolic number in both the visual and auditory modalities: Evidence from fMRI." *NeuroImage* 153 : 16-27.

(chronological)

Hawkins, J., Blakeslee S. (2005). On Intelligence. Published by *Times Books*.

Buzsáki G. (2011). Neural syntax: cell assemblies, synapsembles, and readers. *Neuron*. 2010 Nov 4;68(3):362-85. doi: 10.1016/j.neuron.2010.09.023. PMID: 21040841; PMCID: PMC3005627.

Huth A.G., et al. (2013). A Continuous Semantic Space Describes the Representation of Thousands of Object and Action Categories across the Human Brain. In *Neuron*. 2012 Dec 20; In 76(6): 1210–1224. doi:10.1016/j.neuron.2012.10.014. See figure 2, PPA analysis.

Huyck, Christian R., and Peter J. Passmore. (2013). A review of cell assemblies. *Biological cybernetics*. 107(3). DOI:10.1007/s00422-013-0555-5.

Kruger N., et al. (2013). "Deep Hierarchies in the Primate Visual Cortex: What Can We Learn for Computer Vision?". In *IEEE Transactions on Pattern Analysis and Machine Intelligence*, vol. 35, no. 8, pp. 1847-1871, Aug. 2013. doi: 10.1109/TPAMI.2012.272.

Phillips, Collin, (2013), Derivational Order in Syntax: Evidence and Architectural Consequences. In *Studies in Linguistics* 6: 11-47.

Vogel, Stephan E. (2017) et al. "The left intraparietal sulcus adapts to symbolic number in both the visual and auditory modalities: Evidence from fMRI." *NeuroImage* 153 : 16-27.

Fitch, W.T. (2017). Empirical approaches to the study of language evolution. In *Psychon Bull Rev.* 2017 Feb;24(1):3-33. (2017). doi: 10.3758/s13423-017-1236-5 (2013): 263-288.

Chomsky, N. (2019). Some Puzzling Foundational Issues: The Reading Program. *Catalan Journal of Linguistics*, 263–285. <https://doi.org/10.5565/rev/catjl.287>

Seely, T. Daniel, 2023, [Lecture 1] Working Toward the Strong Interpretation of SMT, https://www.youtube.com/watch?v=vD1Vqv7cmq4&ab_channel=DMCKeioUniversity, 54:30

Chomsky N, Seely TD, Berwick RC, et al. (2023). *Merge and the Strong Minimalist Thesis*. Cambridge University Press; 2023.

Ji, Y., Gamez, D., Huyck, C. (2024). Associative Memory with Biologically-Inspired Cell Assemblies. In: Samsonovich, A.V., Liu, T. (eds) *Biologically Inspired Cognitive Architectures 2023*. BICA 2023. Studies in Computational Intelligence, vol 1130. Springer, Cham. https://doi.org/10.1007/978-3-031-50381-8_43

Del Signore, K. (2024). How Language Could Have Evolved. In: Samsonovich, A.V., Liu, T. (eds) *Biologically Inspired Cognitive Architectures 2023*. BICA 2023. Studies in Computational Intelligence, vol 1130. Springer, Cham. https://doi.org/10.1007/978-3-031-50381-8_27

Del Signore (2024a). Animation of short and long term memory made with c++ prototype. https://www.youtube.com/watch?v=s6flQ1q5i_g

Del Signore (2025). code used to generate figures: <https://github.com/kwd2/graph1/blob/main/stm.C>

Marcolli Matilde, Noam Chomsky and Robert C. Berwick. (2025). Mathematical Structure of Syntactic Merge, August 5, 2025, *The MIT Press*. ISBN: 9780262552523

Pengyu, Liu, et al, (2025). A Survey on fMRI-based Brain Decoding for Reconstructing Multimodal Stimuli, <https://arxiv.org/html/2503.15978v1>