6-24-1997

Link/Loop/Node Networks

Mark E. Gilbert
Western Michigan University, markegilbert@gmail.com

THE CARL AND WINIFRED LEE HONORS COLLEGE

CERTIFICATE OF ORAL EXAMINATION

Mark E. Gilbert, having been admitted to the Carl and Winifred Lee Honors College in 1992, successfully presented the Lee Honors College Thesis on June 24, 1997.

The title of the paper is:

"Link/Loop/Node Networks"

Arthur Falk

Dr. Arthur Falk
Philosophy

Robert Trenary

Dr. Robert Trenary
Computer Science

Donna M. Kaminski

Dr. Donna Kaminski
Computer Science

Link/Loop/Node Networks

Mark E. Gilbe :
Undergraduate Honors Thesis
Lee Honors College
Western Michigan University
Kalamazoo, Michigan, USA

Addendum to *Link/Loop/Node Networks* Section III - The LLN Simulator

This addendum describes the "Blank out Net" feature, as well as a couple of tips for system usage.

First, if the TABLES.MDB is moved out of the current directory, or if the name is changed, the Simulator, on boot up, will put out a dialog form that allows you to tell the Simulator where the LLN Nodes and LLN Links tables are. You can either type in the new path and file name, or press the button with the three dots, and search for the file graphically.

Second, the "Blank out Net" feature on the Main Form will wipe out all nodes and links in the two tables. This will essentially reset the network to its original, blank, state.

Third, you can use these two features to create and maintain multiple networks at the same time. Begin by creating a TEMP directory in the main directory (for instance, C:\LLNS\TEMP). Then, create a set of tables called EMPTY.MDB, and move them to the new TEMP directory. You can do this as follows:

1. Copy a current network's .MDB file
2. Rename it to EMPTY.MDB
3. Move the original network's .MDB file to the TEMP directory
4. Open the Simulator
5. When the form to refresh the links opens, select the EMPTY.MDB file
6. Press the "Blank out net" button.

When this has completed, you will have a blank network to start from. Now, move EMPTY.MDB to the TEMP directory. When you want to create a new network, move whatever file currently has the tables in it to the TEMP directory, and copy the EMPTY file to the current directory, renaming it to whatever you wish. When the Simulator boots up, refresh the links to this table.

The thing to remember about Access' refreshment utility is that if the current tables file is found in the original directory, the form will not appear. It is only when you move the tables file out, or when you rename it or the directory it sits in somehow, that this form appears.

Another thing to keep in mind is the following: this version of the LLNS was written in Access 2.0, the version of Access designed for Windows 3.x systems. As a result, if you have Windows 95, Windows NT, or another later version of Windows, the Simulator cannot make use of the extended file or directory name feature of those later versions. For instance, the following path and file name will not be readable by this version of LLNS:

c:\my documents\temp\lln simulator\data directory\a1.txt

because of the numerous spaces and greater-than-eight-character directory names. To allow the Simulator to continue working, simply keep it in a directory that has names no longer than 8 characters, and restrict the data files and tables files names to the 8.3 format (8 characters, a decimal point, and the 3 letter extension).

Foreword

This document is really two separate documents that have been appended together. The first is *Link/Loop/Node Networks*. This document describes the theory behind link/loop/node networks (LLN), as well as the basic operating instructions for the LLN Simulator (LLNS). The second document is *LLN Simulator Technical Reference*. This second document describes how the LLNS was constructed using Microsoft Access 2.0, as well as how the LLNS simulates LLN.

Link/Loop/Node Networks

Mark E. Gilbert
Undergraduate Honors Thesis
Lee Honors College
Western Michigan University
Kalamazoo, Michigan, USA

Table of Contents

I. Motivation for LLN

The motivation behind Link/Loop/Node Networks (LLN) is simple - it is an attempt at describing how the brain might go about storing knowledge about the world. Specifically, LLN is designed to store **memories** and form **concepts** from those memories. What is it about memories that make them epistemologically interesting topics? A great deal of our explicit knowledge consists of combinations of memories:

> The meaning of the term "memory" can be as general and broad as learning in the widest sense. Every conceivable aspect of learning could be theoretically explained as a direct effect of memory. Memory itself is often thought of as some special ability whose function is to store and retain knowledge for future use. (Furth 1981)

For instance, knowledge of calculus can be reduced (at least in part) to memories of the rules of integration and differentiation. The knowledge of how to fly an airplane can be reduced (again, in part) to the memory that if you pull back on the stick, the nose of the plane will tend to move up.

As time goes by, however, we are prone to forgetting particular details of our memories as well as whole memories. For example, we may forget the actual example of integration by parts that our calculus teacher used to show us how to integrate by parts. However, despite these losses, we are able to retain the ***knowledge*** behind the memory. In other words, we are able to retain the important parts of the memory (the knowledge of integration by parts), but fail to retain the smaller details (the actual example used to demonstrate the technique).

Now, how is it that we are able to forget or lose ***parts*** of our memories, and yet retain the rest? This question deals with the means of storage of our memories: "From a neurobiological point of view, it is widely accepted that memory is an emergent property of the collective behavior of ***systems of neurons*** organized transiently or permanently into functional networks by preferential coupling." (bold italics mine; Laroche, 1995) It would seem logical to assume that if a given memory was stored completely in a single neuron in the brain, then it would be an all-or-nothing situation when it came to losing that memory. In other words, it does not seem possible that you could destroy or damage part of a neuron but still have the rest of the neuron function normally (meaning that the memory stored in that neuron would be partially lost). An alternative to this hypothesis is that memories are not stored in a single neuron, but rather, multiple neurons. In other words, a single memory is distributed over a portion of the brain. This scheme, although more difficult to implement, does have biological evidence supporting it (Patterson and Rose 1992; Vallortigara, Zanforlin, and Compostella 1990), and possess the following advantages:

- The loss of a few neurons here and there would not result in the loss of the entire memory, but rather only in one or more details of that memory.

- Storing memories in this fashion would also mean that specific details of the memory could be more easily analyzed, both with respect to the memory to which they belong, as well as by themselves.

Analyzing the details by themselves would seemingly make it easier for the person to make connections between their memories by comparing and contrasting the details of each. This comparing and contrasting of details of a memory is a big part of the process of "abstraction". Abstraction involves taking memories that have similarities between them and forming a concept based on the memories. For instance, someone can observe several different chairs, and from the qualities that the specific chairs share with each other they can form a concept of a "chair". Now, from there, that person can take the newly formed concept and create higher concepts, such as "furniture", "man-made objects", etc.. In this way, attributes, memories, and concepts will be organized into a hierarchical structure.

The hierarchy described here is really an extension of a line of work done with short-term memory. In particular, this line stems from a 1956 study conducted by George Miller. In this study, Miller hypothesized that in order to remember long lists of items (his actual study used strings of binary digits, 1's and 0's), people grouped, or "chunked" (Miller, 1956) the items, and reduced it in some way. For instance, instead of trying to remember the following string of numbers:

1-0-1-1-0-0-1-1-0-1-1-0

the person could break it up into the following groups:

101 100 110 110

which could then be converted to another base, such as octal (base 8):

5 4 6 6

This shorter string of 4 digits is much easier to remember and manipulate than the original string of 12. This process was termed "chunking" by Miller. Several studies that followed this one showed that people "chunk" in many other areas (Degroot, 1965; Bower and Winzenz, 1969; Johnson, 1972; Chase and Simon, 1973; Chase and Ericsson, 1981).

Although Miller's use of "chunk" was designed to be used to describe the groups of data humans develop to remember things, the term can be expanded to more generally include the attributes of a memory, memories, and the concepts formed from those memories. Concepts become a kind of shorthand for the memories or the other concepts under it, and as a result, allow us to use our knowledge more efficiently, just as chunking allows people to remember (or more generally, manipulate) information.

LLN follows the design specifications laid out here, as the next sections will show.

II. Specifications of LLN

A. Definitions

1. Attributes are features of an object, such as colors, shapes, texture, etc..
2. A node is where an attribute is stored. In each node there is a weight assigned.
3. A link is a connection between two nodes.
4. Loop is a bit of a misnomer - a better term might be "graph". Basically, one loop represents one memory or concept. It is composed of the nodes of the memory/concept, as well as the links that connect them. (The name comes from the look of the original examples drawn to illustrate LLN: several nodes connected in a ring, or loop)
5. The words input vector, vector, and sense data all refer to the same thing: a string of numbers presented to an LLN network to learn.
6. A memory is the loop resulting from an input vector being learned by an LLN network.
7. Abstraction is the process of taking two or more loops, comparing them node by node, and creating a new loop from them. This new loop will contain those nodes that all of the original loops share with each other.
8. Loop A subsumes loop B if A is abstracted from B.
9. A concept is the loop resulting from two or more other loops being abstracted together.
10. Integration is a two step process involving: 1) a memory being created from an input vector; and 2) that memory being abstracted with other memories to form a concept, and then the new concept being abstracted further, etc..
11. Abstraction tree and tree should be taken to mean the conceptual structure of the LLN network, that is to say, the structure of the information stored in the LLN network. The information in LLN is (by design) organized into a hierarchy. This is done for a couple of reasons: it is easier to build and maintain, and it conceptually makes sense - a lot of the information that human beings utilize in their everyday lives can be represented in a hierarchical, or tree-like manner, for example, the classification of plants/animals, the classification of physical objects, etc.. In LLN, the root of the tree is the most abstract concept (at the top), and the most concrete pieces of information - the memories - form the leaves of the tree. All other concepts between the root and the leaves form the limbs or branches of the tree.
12. Querying involves searching the abstraction tree for loops that connect the same set of nodes as what the user specifies.
13. The net manager is the control unit for the network. It controls the formation of new nodes, the creation of new links between nodes, integration of memories into the network, abstraction, and querying.

14. A chunk[1] stands for "memory/concept". Memories and concepts are stored and manipulated in the exact same manner in an LLN network, so it is easier to refer to then using a generic term. For the purposes of LLN, "loop A", "chunk A", and "memory/concept A" are interchangeable phrases.

15. All chunks have a unique chunk id, which is the number assigned to the chunk when it is created. The chunk id's start at 0, and grow up in increments of 1.

16. A level or layer (in this context, the two are interchangeable) stands for the set of all chunks that have been abstracted by the same amount. For example, two memories would be on level 0 (the lowest level) since they both come from the sensory apparatus directly. The concept is abstracted from those two memories would be on level 1 because this concept would be once removed from the system's senses, and so on up the abstraction hierarchy. All memories are found on level 0, and all concepts are found on levels 1 to n-1, where n is the number of levels total in the abstraction tree (and as a result would contain the most abstract concepts).

17. Closeness describes the amount that two chunks differ from one another. This is measured in terms of the weights of the nodes of the two chunks, and is unitless.

18. Vigilance is a unitless number that is used to determine if two chunks are close enough to one another to be abstracted together into a concept. If the closeness measure for a pair of chunks meets or exceeds the vigilance setting, they are considered close enough. If the closeness measure does not meet or exceed the vigilance, the two chunks are not close enough. Increasing the vigilance setting will tend towards fewer concepts being formed from the memories, and the concepts that are formed will tend to contain concepts that are more similar to one another. Decreasing the vigilance setting will tend towards more concepts being formed from the memories, and the concepts that are formed will tend to contain concepts that are less similar to one another.

    The vigilance setting is always between 0 and 1. It remains constant for all layers in the abstraction tree for the following reason: as you move up the tree, the numbers of nodes total between chunks being compared to one another will decrease, as will the number of similar nodes. This provides a general rule of thumb that allows an outside source (i.e., a person manually analyzing the network) the ability to tell a concept apart from a memory, or a chunk at level i from a chunk at level i+1.

    The following assumption is made concerning the closeness of chunks in different layers of the tree: chunks that are close initially will remain close throughout the layers, and the far ones will remain far. Making this assumption simplifies the algorithms.

---

[1] It is a coincidence that (Miller 1956) and LLN both use this term. Although they have related meanings, their definitions were arrived at independently of one another, and should not be taken to be interchangeable.

15. Deterioration refers to the removal (analogous to biological death) of a node from the system, the removal of an LLN link from the system, or the unauthorized modification of a weight or other information stored in a node in the network ("unauthorized" here means not occurring as a result of the normal operation of an LLN algorithm).

16. Response refers to an LLN node storing an attribute that matches one of the attributes in the input vector being presented to the network.

17. The category and category position of an attribute serve to identify it, and distinguish it from other attributes. They can be thought of as being analogous to (X,Y) coordinates. The motivation for these two is biologic: a "category" is analogous to a type of sensory input (i.e., a sound, a smell, a sight, etc.), while a "category position" is analogous to refinements in the type of sensory input (i.e., pitch, tone, and clarity for sound; heat and texture for touch, etc.).

B. LLN algorithms and operation

LLN's operation can be broken down into three pieces as follows: New Memory Addition, Abstraction, and Tree Querying.

New Memory Addition

For the sake of argument, let us suppose that there is some sort of sensory apparatus feeding an LLN network information. One of the first questions that you can ask is: What does this information look like? In this first generation of LLN, the sense data will be strings of base 10 numbers representing the objects' attributes.

The net manager begins by looking at the first attribute in the input vector. The set of the LLN nodes that respond to the attribute are collected in a list, and set aside for the moment. Then, the algorithm moves on to the next attribute, and so on, until a list of responded LLN nodes has been compiled. Now, there may be multiple nodes in the list that responded to the same attribute. The reason for this would be deterioration of the structural units of the network. To deal with this scenario, the first duplicated LLN node will be used, and the rest ignored. The reason for this is simple - if there are multiple LLN nodes that now respond to a particular attribute, then there is no way to tell which LLN node corresponds to the "real" attribute.

If the attribute does not cause any LLN nodes to respond (meaning this attribute either never existed in the network to begin with, or no longer exists as a result of deterioration), then it must be created. The creation of a new attribute requires the allocation of a node (either through the allocation of more RAM or from an available node pool). Once the new node has been created, it is added to the set being compiled for this chunk, and the net manager takes care of the rest of the attributes for this memory. When it has finished, the net manager will use the list of nodes that it has compiled to perform the abstraction process.

Once all the nodes have been found and/or created, the net manager sets up links to the nodes. The number of links created, the structure of the graph set up, and the mechanics involved in the actual "linking" of the nodes are all highly implementation dependent. The first generation LLN will

create a complete graph between the nodes of a memory, meaning that every node in the memory will be connected to every other node in the memory.

After the links have been established the next step is the abstraction process.

Abstraction

The process of taking a memory and abstracting it to complete the memory's integration into the system is needed to give the information stored in the net conceptual structure. The basic idea here is to find other memories that are similar to the one being integrated. Then, with that new concept (either newly created or newly modified) the net manager tries to create/modify other concepts that are conceptually above the new one.

The first order of business is to find the memory or memories which will be abstracted with the new one. The net manager begins the search for candidate memories by visiting all nodes belonging to the new memory. It records all memories that connect to each node that are on the same level as the new memory (level 0 - remember, concepts at levels higher than level 0 may use this same node, so we must distinguish between them). Once it has visited all of the new memory's nodes, it then begins a traversal of nodes of the memories in the set it just compiled. Once it has visited all the nodes in a memory, the closeness ratio for that memory to the new one is calculated. This calculation is given below:

Formula II-1

$$\text{ratio} = \frac{X}{X + \text{sum of the averages of the weights of the corresponding } \textbf{non-shared} \text{ nodes of the two memories}}$$

where X = sum of the weights of the nodes that are shared between the two memories, and "corresponding" refers to two units, one from each memory, that are in the same category and category position as the other.

This ratio is then compared to the vigilance. If the ratio exceeds the vigilance, then the new memory is "close enough" to the existing ones, and it is left in the set. If it does not exceed the vigilance (i.e., if the ratio is less than or equal to the vigilance), then the new memory isn't close enough to the existing memory at hand. That memory is removed from the list, and the net manager moves on and looks at the next memory in the set.

When these calculations are done, the memories that are left in the set are the ones that are the best candidates for being the memories to abstract with. Now, there are several cases that we must look at:

1. There are no memories in the set. The abstraction process stops here. If no match is found for a new chunk to abstract with, then it means that the new memory is so radically different from the rest of the memories in the network that it doesn't need to be abstracted. Remember, there is one main reason that we are performing abstraction on the memories: to show the relationships of one memory to another. A new memory that is far enough away from everything else in the network will be

able to stand out enough that a search engine will be able to find it easily. An example of this would be if the network only had experiences of green spherical or elliptical objects, and you present to it a red cube. The cube is unlike anything that it has ever experienced to date, and as a result would not have anything in the network that was like it. Finding the memory of it would be easy: just look for something that has edges and is red - the only thing that will match this description will be the cube. It would seem logical that this will occur a lot when the network first begins to acquire information, and would become increasingly unlikely as the network learns more and more about the objects and concepts of its world.

2. There is exactly one memory in the set, and it does not already have a concept that it is subsumed under, create a new concept subsuming it and the new memory.
3. There is exactly one memory in the set, and it already has a concept to which it belongs.
4. There is more than one memory in the set, and some or all of them may have a concept to which they already belong. In this case, select the memory that has the highest weighted ratio and abstract the new memory with it (creating a new concept if the memory is not already subsumed y one, and altering its subsuming concept if it is subsumed by one). If two or more have an identical weighted ratio (rare but still possible), then choose the one with the lowest chunk id, and abstract the new memory with this one by creating a new concept (if the memory chosen is not subsumed by a concept) or alter the existing concept (if the memory chosen is subsumed by a concept).

   The choice of the lowest chunk id over all others is an arbitrary one. If there are two or more chunks that are equally close to a third, there is no logical way to choose which of the two to abstract the third with. All three of them cannot be abstracted together because the memories that are in the set may not be close enough to each other to form a concept (if they were they would have been abstracted together at an earlier time).

In truth, cases 1-3 are really special cases of number 4, but they are enumerated here to make the logic behind this portion of the algorithm explicit.

As a side note, the net manager will be able to tell if a particular chunk has been subsumed by another chunk by using a pointer stored in the lower chunk pointing to the next higher chunk. In other words, when a new concept is created to subsume a set of chunks, all of the links of the subsumed chunks will get a pointer set to the new chunk's id. If a new chunk is being integrated into the system under a current concept, then the new chunk will get its link pointers set to the existing concept. This method of setting a pointer from a particular chunk to the chunk that it is subsumed by may seem to be cheating, but the hard part of the process has already been accomplished at this point by the system; that is figuring out which higher chunk the lower one should belong under. A pointer system will enable the querying of the network to proceed faster.

Once the proper memory has been chosen as well as the technique to integrate the new memory in, then all that is left is to implement the technique. The two possible techniques are creation (of a new concept) or alteration (of an existing concept).

The creation of a new concept entails figuring out which nodes are shared between the memories being abstracted. These shared nodes will become the nodes that make up the concept. To

select this set of nodes, traverse the nodes of one of the memories and see which ones the other memory shares. Establish a new set of node links with this set of nodes, and alter the weights of the nodes in this new concept. The formula for this weight alteration is given below:

Formula II-2

$$\text{new weight} = \text{old weight} + \left(\frac{\#\text{ nodes in concept}}{\text{number of nodes in memory}} * 0.01\right)$$

Alteration of an already existing concept involves increasing the nodes' weights that are shared between the existing and new memories and are already in the concept that will subsume both; in both levels, these sets of nodes to be altered are identical to one another - this is guaranteed by the way that the existing memory was selected (i.e., through calculation of the weighted ratio).

At this point, the new memory has been successfully integrated into the system. The abstraction process begins all over again for the next layer if there has been a change made to the level above the memory (i.e., if a change has been made to level 0, the next level to look at is level 1), such as the creation or alteration of a concept. The process for doing this for the new concept is identical to the one just described.

As an observation, new attributes (i.e., new nodes) are never created when abstraction occurs, only when a new memory is learned. Also, LLN has been designed so that a chunk may only integrate with other chunks on the same level as itself.

Tree Querying

The querying algorithms described here are designed to provide little more than information concerning network integrity, i.e., how well the information originally stored in the network has been retained despite possible deterioration.

There are two types of querying that can be performed on this version of LLN. The first is a search for any chunks similar to the user-specified chunk, and the second is a complete chunk-map. The former method will find any chunks that connect all of the nodes that the user specifies. The latter will map out in a hierarchical form the chunks that have been learned by the system according to the level that each chunk is on, as well as by which chunk (if any) subsumes it.

The first thing that must be accomplished in querying the tree to see if a particular chunk is present is to locate the LLN nodes that represent the attributes of the chunk to be searched. Now, this first part may seem very time consuming. There are a couple of methods for reducing the number of node searches that must be performed. The first would be to spread the search across multiple processors. Another approach (in lieu of a multiprocessing machine) would be to search for the attributes according to category. In other words, look for an attribute in category 1 in the nodes that store category 1 nodes, look for category 2 attributes in category 2 nodes, etc..

If there is a particular node that the user requested, but that did not exist in the system, quit - either that node has been lost (destroyed) since it was learned, or it was never learned in the first place. If all of the user-requested nodes are present, continue. Now the search for the chunks begins. Go through the list of nodes, and see what other nodes are connected to them. This can be achieved by looking at the links connected to the node in the following manner:

If the link is already activated, then move on to the next link.

If the link is not already activated, follow that link and see if it connects the current node to another node in the list compiled above. If it does, activate that link (this could be little more than setting a Boolean value in the link). If it doesn't, leave the link alone. In either case, move on to the next link.

If all links for a particular node have been examined, move on to the next node. Continue this procedure until all nodes have been examined.

Once the above loop has been completed, the system will have a set of activated nodes (the ones compiled in the first part of this algorithm), and a set of activated links. The next portion of code will operate on these two sets.

Walk through the set of nodes. From each of those nodes, follow the links for a particular chunk (each link will have a value holder of some sort that describes which chunk it belongs to) that have been activated to the other activated nodes. Continue this search of links until all of the links in the set for this particular chunk have been found. Compare the set of nodes made up by this chunk with the set that the user requested. If the former has all of the latter, keep the chunk, and move on to the next chunk. If the former does not have all of the nodes from the latter set, deactivate all of the links that that chunk is represented by.

At this point, a couple of checks are required. For a particular node - will the elimination of this chunk's links to it mean that this node will have no activated links to it anymore? If so, then exit - this scenario means that there are no chunks that link all of the user-specified attributes. If not, then have all of the nodes and links (and therefore, chunks) been examined? If so, exit. If not, move on to the next chunk. When this weeding out of chunks finishes, the chunk(s) that remain in the list will be presented to the user. Now, a *list* of chunks returned doesn't initially seem to be useful; however, it will, at the very least, tell the user that the information provided to search with was ambiguous.

The second querying technique, the chunk map, is designed for system analysis, and would not be utilized as a normal network accessing method for a couple of reasons:

1. The chunk map basically lays out everything that the network has learned, along with that information's organization. There does not seem to be a need for a function of this sort if the network was being used by an inference engine or other AI system.
2. This function would be fairly complicated to implement by merely following links and nodes, and as a result, requires shortcuts to the logic to improve the performance. Such shortcuts are implementation dependent, and would require access to the links and nodes via methods other than through nodes and links, respectively.

III. The LLN Simulator

The LLN Simulator is the implementation of the version of LLN discussed in this paper. It was implemented using Microsoft Access 2.0. This Simulator is appropriately named, as the algorithms that the code follows are not identical to the algorithms described here - the results are the same however. For a more detailed description of the implementation details of the Simulator, please refer to the *LLN Simulator Technical Reference*. This manual describes the algorithms used, the coding specifics, and a section describing how the algorithms in the Simulator are equivalent to the ones in this description.

The Simulator is very easy to use. Every form in the system has the same basic design. Going along the bottom of the forms in the royal blue strip are a set of hyperlinks. The possible choices are: Exit Database, Main Menu, Train Memories, and Query Network. Pressing the first will exit the database and Access. Pressing the second will return the user to the Main Menu. Pressing the third will take the user to the Train Memories form. Pressing the fourth will take the user to the Query Network form. Not all hyperlinks are available from all forms, e.g., if you are on the Train Memories form, the Train Memories hyperlink is not present since you are already there. Also, the hyperlinks that are colored yellow (as opposed to the light blue) will take the user to the next higher menu in the menu structure.

On boot-up, the Main Menu allows the user to perform one of two sets of functions: Train Memories or Query Network. Pressing the first button will take the user to the Train Memories form. Pressing the second button will take the user to the Query Network form.

On the Train Memories form, the user will find two text boxes where something may be typed (sunk into the form, colored white), two command buttons (raised, colored light gray with blue lettering), and two labels (raised, light blue lettering with a royal blue background) which describe the usage of the other controls on the form. The first text box allows the user to enter a single memory at a time according to a fixed format as described by the label for this box. Once the user has the memory entered, pressing the "Train this memory now" button to the right will train that memory in. If the user has several memories that they want trained in, they can enter them into a text file in the following format:

i
$Memory_1$
$Memory_2$
.
.
.
$Memory_i$

Where i is the number of memories in the file, and the memories entered are in the same format as described by the first label (see above). Once this file has been created, the user can type the filename (and path if needed; if the user only types the file's name, the code will look in the current directory for it only) in the second text box on this form, and press the "Load and Train now" button. The code will open the file, read in the memories, and train each one just as if the user entered them one by one in the first text box.

If the user goes to the Query Network form, they can either type a chunk to search for (the system is capable of search for not only memories, but the concepts that have been abstracted from them) in the text box, and press "Search for Chunk" button to the right of it. Once the routine has completed, a report is opened that will list out the chunks that contain all of the attributes the user entered. The user can also press the "Run Chunk Mapping Report" button, which will, as the button says, run the chunk mapping report and display it on screen. Both of these reports can then be printed.

A typical run-through of the system would be as follows:

- The user types in a set of memories into a file, and then begins the Simulator.
- The user goes to the Train Memories form, and enters the full path and file name of the memory file created in the above step into the second text box. They then press the "Load and Train now" button.
- Once the memories have been trained in, the user goes to the Query Network form, and runs out the Chunk Mapping Report. They look to see what concepts have been formed from the memories they presented to the system.
- The user then goes back to the Train Memories form, and enters another new memory, this time using the first combo box. They then press the "Train this memory now" button to train it in.
- Once this memory has been trained in, the user goes back to the Query Network form, and searches for the concept that that concept should have trained into using the text box and the first button. Once this report has been run, they print it, and the press the Exit Database hyperlink to shut the Simulator down.

A note about the Simulator - the tables that the chunks are stored in are kept in a separate Access database called TABLES.MDB. The Simulator itself is stored in the LLN.MDB file. The latter attaches to the former's tables. When you exit the Simulator, the copy of the TABLES.MDB that you stored will automatically get any unsaved information saved. Breaking the data apart from the code in this manner has these advantages:

1. The Simulator can be upgraded fairly easily without requiring the alteration of the tables that the data is stored in.
2. If you want to have multiple LLN networks in existence, you simply need to keep separate TABLES.MDB files, one for each network - i.e., you do not need multiple copies of the LLN.MDB file. The files that contain the data do not even have to be named "TABLES.MDB". They can be renamed anything the operating system will allow (so long as the ..MDB file extension is kept). When you want to switch between networks, simple refresh the attachments from one database to the new one.

Breaking the tables apart from the rest of the Simulator, however, has one major drawback: if you move the TABLES.MDB file out of the directory that it was in, the LLN.MDB will need to have the attachments refreshed to the new location since Access stores the complete path and file name of the TABLES.MDB file.

IV. Hypotheses

There are two main areas of LLN that will be tested: 1) the storage and retention of the network during the learning of memories and formation of concepts, and 2)the integrity of the information stored in the network before and after portions of the network structure are removed or changed.

For the first area, dozens of sample memories will be presented to the network for it to learn. From there, the network will attempt to abstract them into concepts, and those concepts into higher concepts, etc.. Then, the network will be queried in an attempt to retrieve the memories that have been stored, as well as the concepts that have been formed. There will be three main things being sought in the data: 1) have the pre-existing memories been preserved in the network as the new ones are added? 2) have the pre-existing abstractions been preserved in the network as the new memories are added and the new concepts formed? and 3) are the abstractions formed logical?

For the second area, random structural units will be removed from the system, as well as random weights will be altered. Select affected and non-affected memories and concepts will attempt to be accessed. What is being sought in the data is the following: is the network still able to function given the deterioration that has occurred. Deterioration should lead to longer search or traversal times (as pieces of the network would be missing, causing the search/traversal algorithms to back up and come in another way), as well as causing pieces of chunks to be irretrievably lost. It will also lead to multiple nodes responding to a particular attribute being presented to it which will also lead to longer search times, as well as the possibility that the original attribute stored in a deteriorated node to become irretrievably lost.

The following input vector sets will be used during the test phase of LLN:

A1	A2	A3	B1	B2	B3
0;1;2;3;6;	0;1;2;3;6;	1;3;1;2;0;4;	4;4,5;6,4;3;2;	2;4;4;6,4;3;2;	2;4;4;6,4;3;2;
1;3,40,5;1;2;2;2;	0;1;1;2;3;	0;1;1;2;3;	2;4;4;6,4;3;2;	2;3;3;6;3;	2;3;3;6;3;
0;1;1;2;3;	4;66,1;15;;4;	1;1;2;3;6;	1;2;3;3,4;5;7;	2;5;4;6,4;3;	2;3;4;6;3;;
4;66,1;15;;4;	1;1;2;3;6;	1;2,40,5;1;2;3;	7;2;3,3;4,5;6;	1;2;3;3,4;5;7;	2;5;4;6,4;3;
1;1;2;3;6;	1;2,40,5;1;2;3;	1;0;1;2;4;	1;0;0,1,2;4;4;	7;2;3,3;5,5;6;	1;2;3;3,4;5;7;
1;2,40,5;1;2;3;	1;0;1;2;4;	4;60,1;4;4;	2;3;4;6;3;;	7;2;3,3;4,5;6;	4;4,5;6,4;3;2;
0;2;2;3;6;	1;3,40,5;1;2;2;2;	0;1;2;3;6;	1;1;0;1;2;3,3;4;	4;4,5;6,4;3;2;	2;4,1;4;6,1;3;2;
0;1;1;2;4;	0;2;2;3;6;	2;1;2;3;6;7;	4;4,5;6,4;3;4;2;	2,1;1;1;4;4;	2,1;1;1;4;4;
4;60,1;4;4;	1;3;;2;0;4;	1;3,40,5;1;2;2;2;	1,1;2;1;4;4;	2;3;4;6;3;;	1;0;0,1,2;4;4;
0,1;1;2;3;3;5;	1;2;1;2;2;	0;2;2;3;6;	2;4,1;4;6,1;3;2;	1;1;0;1;2;3,3;4;	7;2;3,3;5,5;6;
0;1;2;3;4;	4;60,1;4;4;	1;3;;2;0;4;	2;1;4;6;3;;1;	1,1;2;1;4;4;	1;1;0;1;2;3,3;4;
4;3;3;3;4;	0;1;2;3;4;	0;1;2;3;4;	0;1;0,1,2;3,1;	0;1;0,1,2;3,1;	0;1;0,1,2;3,1;
7;1;2,3;4;5;	4;3;3;3;4;	4;3;3;3;4;	2,1;1;1;4;4;	1;0;0,1,2;4;4;	2;4;4;6,4;
7,1;2;3;3;6;	7;1;2,3;4;5;	7;1;2,3;4;5;	7;2;3,3;5,5;6;	2;4;4;6,4;	1,1;2;1;4;4;
2;1;2;3;6;7;	7,1;2;3;3;6;	1;2;1;2;2;	2;3;3;6;3;	4;4,5;6,4;3;4;2;	4;4,5;6,4;3;4;2;
0;2;1;2;4;	2;1;2;3;6;7;	0,1;1;2;3;3;5;	2;5;4;6,4;3;	2;4,1;4;6,1;3;2;	7;2;3,3;4,5;6;
1;0;1;2;4;	0;2;1;2;4;	4;66,1;15;;4;	2;4;4;6,4;	2;1;4;6;3;;1;	2;1;4;6;3;;1;
1;2;1;2;2;	0,1;1;2;3;3;5;	7,1;2;3;3;6;			
1;3;1;2;0;4;	0;1;1;2;4;	0;2;1;2;4;			
1;3;;2;0;4;	1;3;1;2;0;4;	0;1;1;2;4;			

The following seven hypotheses have been derived from the areas of investigation denoted above. Beneath each hypothesis is the corresponding null hypothesis. Beneath the null hypothesis are the instructions for creating the networks that are intended to disprove the corresponding null hypotheses. In the following instructions, "CMR" refers to the "Chunk Mapping Report" of the LLN Simulator.

Hypothesis 1 - *A node of a chunk will be accessible until all links to that node are removed from the system.*

Null Hyp: A node of a chunk will be accessible when and after all links to that node are removed from the system.

1. Use **a1.txt** as input file.
2. Remove 1 link to node ***i*** for each of two different chunks in the network. This will be network **a1h1a**. Run CMR.
3. Copy network **a1h1a**. Remove all but one link for node ***i*** for the same two chunks. This will be network **a1h1b**. Run CMR.
4. Copy network **a1h1b**. Remove last link for node ***i*** for the same two chunks. This will be network **a1h1c**. Run CMR.

Hypothesis 2 - *Once all links to node* **i** *of chunk* **j** *are removed, searching for chunk* **j** *using node* **i** *in the search string will result in chunk* **j** *not showing up in the results.*

Null Hyp: Once all links to node **i** of chunk **j** are removed, searching for chunk **j** using node **i** in the search string will result in chunk **j** showing up in the results.

1. Use **a1h1a, a1h1b,** and **a1h1c** as the base networks.
2. Perform a manual query on the two affected chunks in the networks, and report if they showed up in the search results or not.

Hypothesis 3 - *Deleting node* **i** *will result in all chunks connected to node* **i** *not being able to access it.*

Null Hyp: Deleting node **i** will result in all chunks connected to node **i** still being able to access it.

1. Use **b1.txt** as the input files.
2. Create a network called **b1h3a** from the input file. Then, remove a node, and run CMR.

Hypothesis 4 - *Removing links for chunk* **i** *won't affect its abstraction with other chunks until all links for a given node for chunk* **i** *are removed.*

Null Hyp: Removing links for chunk **i** won't affect its abstraction with other chunks when and after all links for a given node for chunk **i** are removed.

1. Use **a1h1a, a1h1b,** and **a1h1c** as the base networks.
2. Create new, analogous, networks called **a1h4a, a1h4b,** and **a1h4c**. These networks will be identical to their base networks, except a chunk that would normally abstract with the two affected chunks will be added.
3. Run the CMR for each of the three new networks.

Hypothesis 5 - *This is a corollary hypothesis to Hypothesis 4. Removing all links connecting to node* **i** *for chunk* **j** *may result in another chunk* **k** *that did not previously abstract with chunk* **j** *to abstract with chunk* **j.**

This hypothesis will remain untested, although explained. It should be rather intuitive given Hypothesis 4.

Hypothesis 6 - *The order that the input vectors are presented to the network may affect the final chunk structure.*

Null Hyp: Since this hypothesis only claims the order *may* affect the final structure, it will be enough to show an example where it does affect the structure, and an example where it doesn't affect it.

1. Use **a1.txt, a2.txt, a3.txt, b1.txt, b2.txt,** and **b3.txt** as the input files.
2. Create networks **a1h6, a2h6, a3h6, b1h6, b2h6,** and **b3h6** from these input files.
3. Run the CMR for each of the networks.

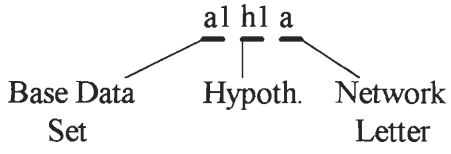
Hypothesis 7 - *As the network learns, a definition for an object represented by a chunk will be derived from the more heavily weighted nodes of that chunk.*

This hypothesis will not be tested in this project. However, it would seem logical that the definition of an object would be highly dependent on the following:

1. The time at which you requested the definition with respect to the network's training.
2. The order the input vectors were presented to the network.

V. Results

If a particular hypothesis' test required CMRs to be run for it, those CMRs will be found attached to the end of this document. They are marked in the upper right corner with a code in the following format:



If there happens to be more than one network created from the same base data set, and for the same hypothesis, they are given a unique letter designation, beginning with "a".

Hypothesis 1

Chunks 2 and 4 had all of their nodes accessible until all links to nodes 1 and 6 respectively were removed. Then, as shown in a1h1c, one of the original nodes is missing. It is more significant to note that all links had to be removed before the node was lost - there was no gradual decay of performance. However, the network was certainly fault tolerant.

These results show the null hypothesis here to be wrong, and therefore the original hypothesis is correct.

Hypothesis 2

Performing manual queries for chunks 2 and 4 in networks a1h1a - a1h1c had similar results as were found for Hypothesis 1: searches for the original data sets for chunks 2 and 4 were successful in a1h1a and a1h1b, but not in a1h1c.

These results show the null hypothesis here to be wrong, and therefore the original hypothesis is correct.

Hypothesis 3

Removing node 15 (represents a "2" in category 2, and category position 1) resulted in chunks 2, 3, 17, and 18 not being able to access it.

These results show the null hypothesis here to be wrong, and therefore the original hypothesis is correct.

Hypothesis 4

The input vector "1;1;2;3;7;" abstracted with chunk 4 in a1h4a and a1h4b, but not in a1h4c. "0;1;1;2;5;" abstracted with chunk 2 in the first two networks, but not in a1h4c. In a1h4c, it abstracted with chunk 8.

The latter of these two tests is not as clear as the former because chunks 2 and 8 are so similar to one another.

These results show the null hypothesis here to be wrong, and therefore the original hypothesis is correct.

Hypothesis 5

This hypothesis was not tested in this project, but the support for this hypothesis is as follows: Suppose you have the following two input vectors: "1;1;2;3;7;8;" and "1;2;2;3;7;". Assume a vigilance setting of .8, and assume that these are the very first two vectors that you present to a brand new network (making this last assumption means the node weights are all 1). Now, if these two vectors are presented in the above order, one right after another, and before any deterioration occurs to the links/nodes, the following calculation holds:

Closeness = 5/6.5 = .76

With the above closeness calculation, the two memories would not abstract together.

Now, let's say that you present the first vector, allow it to have a memory created for it, and then removed all of the links to the 8 in category 6, category position 1. The first memory is thus reduced to "1;1;2;3;7;". Now, let's say that you now present the second vector to the network to learn. Assuming a vigilance of .8 again, the closeness calculation is as follows:

Closeness = 4/5 = .80

This closeness is enough to allow the two vectors to be abstracted together.

Hypothesis 6

The following three networks demonstrate that order does not matter to the final tree structure: a1h6, a2h6, and a3h6. The following three networks demonstrate that order *does* matter to the final tree structure: b1h6, b2h6, and b3h6.

Given this data, the hypothesis is supported.

Hypothesis 7

This hypothesis was not tested in this project.

VI. Conclusions

The system has fulfilled the original design requirements in the following ways:

1. The system seems to be fault tolerant for the following reasons:
   a) Removing a few links from a chunk doesn't affect the chunks at all.
   b) Removing a node doesn't result in the entire chunk being lost.

2. The system has also demonstrated the ability to abstract the input vectors into logical concepts. This is evident through examination of the CMRs.

Despite these accomplishments, the system does have some serious limitations in performance and in power. These are outlined below, and serve as a basis for improvements in the system.

1. The system's access to the stored information is slow, partly due to LLN being implemented in software (as opposed to hardware), but mostly due to the algorithms themselves.

2. The system cannot handle time at all. For example, let's say that the system were to learn what Joe looked like today. If Joe were to get a haircut, and have the system "sense" him again, it would "think" that it was seeing a being that was very similar to a previously stored memory, but one feature was different - Joe's hair. It would attempt to then create a brand new memory, and since the new memory of Joe and the old one of Joe would be very close to being identical, it would then try to form a concept of "Joe". However, this does not seem to be the way that we, as humans, do it. A person asked to perform the same task as the system would simply take note that Joe had gotten a haircut - they would deny that the old Joe and the new Joe were two very similar, but *distinct* people.

   This problem is more the responsibility of a reasoning unit that is able to manipulate the information that is stored in the network. Such a unit could realize that a new memory that was very close to a previously stored one does, in fact, refer to the same object in the world. As a result, it would direct the LLN network to modify the already existing memory to fit the new information. Furthermore, it may also store (somehow) the change that was made. In this way, the network could be made to recall what the object looked like in the past *and* in the present. This storing of the change made runs into a similar problem as before - the system does not as of yet have mechanism for storing time. In this situation, where the system is directed to store the change made, it could store the time and day that the change took place, and therefore when queried about the past, it would take into account the time and day being asked about, and from there could weed out the memories that did not apply to times after that one.

3. Also, in a related strain as the previous note, the system is unable to store what can be termed as a "scenario". A "scenario" is a time-dependent situation in the real world, for example, a debate. The system has no easy way of encoding what person A said, and how person B responded. This above constraint is a chief reason that the system is unable to get

a grasp on much more abstract concepts, such as justice. In order to form such an abstract concept, the system would need to observe situations where justice was served and where it wasn't before it could learn what "justice" itself was. It does not seem likely or logical that it could learn this by only being presented with objects of the world.

An objection that may be raised is that the system is merely manipulating symbols, and really has no understanding what those symbols may mean or refer to in the real world. The response is that this system it is not supposed to place any particular meaning to the nodes or chunks it creates; rather this system is designed to be a means of storage only. It has been designed to allow for the addition of other modules to be added onto it, such as an inference engine, more sophisticated sensory devices, etc.. Once these other modules are in place, this objection will have much more bearing on the system as a whole, and it will be dealt with at that time.

Despite these limitations, the system provides a good platform from which more advanced versions can be built, and improvements made. Below is a list of such possible improvements to the current version of LLN, and the LLN Simulator:

1. A complexity (time and space) analysis should be done in the attempt to make the algorithms more efficient.
2. Implement a way for the system to add an attribute to an existing chunk, thereby allowing the system to learn more about objects it has already sensed.
3. Implement the LLN nodes as neural networks, such as back propagation or ART, etc., in the attempt to further boost the fault tolerance of the network.
4. Develop methods for traversing the network that are more connectionistic/analog in nature in an attempt to improve performance.
5. Add in to the LLNS a more user-friendly way for the user to switch between various LLN networks, or even to create a blank one to start with.
6. Develop a way for the system to handle time. This may be in the interpretation of the chunks stored in the network, where the interpretation would be performed by an external module.
7. Develop an interface unit that would bring with it a set of primitives that would work on an LLN network.
8. Explore an alternative "link" design as suggested by Dr. Trenary*. Instead of creating multiple links when a set of memories and concepts use the same two nodes (i.e., create one link for each memory and one for each concept that used that pair of nodes), create a single link that had a weight attached to it. Then, every time a new memory or concept, the weight is increased to reflect the added usage. Now, to simulate decay of the system, this weight could then be diminished by some degradation formula. What this allows is a much more smoother degradation that simply removing an entire link. The difficulty with this is the new interpretation of the links.
9. Explore another alternative method of linking the nodes together as suggested by Dr. Trenary and Dr. Kaminski*. Instead of trying to find a unilateral formula for the number of links created, why not link the nodes of one category to the nodes to another category

---

* Dr. Arthur Falk, Dr. Robert Trenary, and Dr. Donna Kaminski formed the thesis committee that oversaw this project. Their help and input were invaluable, and greatly appreciated.

depending on how important those categories are towards the memory at hand. For instance, if we know that categories 1 and 3 are very important for a particular memory, we will connect the nodes between those two categories in a connected graph of links. However, categories 2 and 4 are not as crucial to the memory, so those nodes would be linked to the others using less than a complete graph of links. The question arises - how would the system know what categories are more crucial to a particular memory BEFORE it learns that memory?

10. Explore another degradation technique as suggested by Dr. Kaminski. If the system were to determine which nodes are seldom used by the memories and concepts stored in the network, those nodes could be weeded out, and used to store other, new nodes.

Bibliography

Bower, G.H. and D. Winzenz. "Group Structure, Coding, and Memory for Digit Series". Journal of Experimental Psychology" May 1969: 1-17.

Chase, W.G. and K. A. Ericsson. "Skilled Memory". Cognitive Skills and Their Acquisition. Ed. J.R. Anderson. Hillsdale: Erlbaum, 1981.

Chase, W.G. and H.A. Simon. "Perception in Chess". Cognitive Psychology 4 (1973): 55-81.

DeGroot, A.D. Thought and Choice in Chess. Mouton: The Haugue, 1965.

Furth, Hans G. Piaget and Knowledge, 2nd Edition. Chicago: The University of Chicago Press, 1981.

Gilbert, Mark E. LLN Simulator Technical Reference. Kalamazoo, 1997

Johnson, N. E. "Organization and the Concept of a Memory Code". Coding Processes in Human Memory. Eds. A. W. Melton and E. Martin. Washington: Winston, 1972.

Laroche, Serge, et al. "Neural Mechanisms of Associative Memory: Role of Long-Term Potentiation". Brain and Memory: Modulation and Mediation of Neuroplasticity. Ed. James L. McGaugh, Norman M. Weinberger, and Gary Lynch. New York: Oxford University Press, 1995: 277-302.

Miller, George. "The Magic Number Seven Plus of Minus Two: Some Limits on Our Capacity for Processing Information". Psychological Review. 63 (1956): 81-97.

Patterson, Teresa A. and Steven P. R. Rose. "Memory in the Chick: Multiple Cues, Distinct Brain Locations". Behavioral Neuroscience. 106 (1992): 465-470.

Vallortigara, Giorgio, Mario Zanforlin, and Silvia Compostella. "Perceptual Organization in Animal Learning: Cues or Objects?" Ethology. 85 (1990): 89-102.

Raw Data

This section contains the Chunk Mapping Reports (CMRs) for the networks that were created for the hypotheses laid out in Section IV.. A CMR describe all memories and concepts stored in a particular network. The CMRs are arranged in ascending order of the hypothesis the were created for. The labeling for the CMRs is described in Section V..

Chunk Mapping Report

;1;2;3;6; - 5
0;1;2;3;6; - 0
1;1;2;3;6; - 4
0;2;2;3;6; - 7
0;1;2;3;4; - 12
2;1;2;3;6;7; - 16

0;1;1;2; - 9
0;1;1;2;3; - 2
0;1;1;2;4; - 8

1;3;;2;0;4; - 22
1;3;1;2;0;4; - 20
1;3;;2;0;4; - 21

1;3,40,5;1;2;2;2; - 1

4;66,1;15;;4; - 3

1;2,40,5;1;2;3; - 6

4;60,1;4;4; - 10

0,1;1;2;3;3;5; - 11

4;3;3;3;4; - 13

7;1;2,3;4;5; - 14

7,1;2;3;3;6; - 15

0;2;1;2;4; - 17

1;0;1;2;4; - 18

1;2;1;2;2; - 19

Chunk Mapping Report

;1;2;3;6; - 5
0;1;2;3;6; - 0
1;1;2;3;6; - 4
0;2;2;3;6; - 7
0;1;2;3;4; - 12
2;1;2;3;6;7; - 16

0;1;1;2; - 9
0;1;1;2;3; - 2
0;1;1;2;4; - 8

1;3;;2;0;4; - 22
1;3;1;2;0;4; - 20
1;3;;2;0;4; - 21

1;3,40,5;1;2;2;2; - 1

4;66,1;15;;4; - 3

1;2,40,5;1;2;3; - 6

4;60,1;4;4; - 10

0,1;1;2;3;3;5; - 11

4;3;3;3;4; - 13

7;1;2,3;4;5; - 14

7,1;2;3;3;6; - 15

0;2;1;2;4; - 17

1;0;1;2;4; - 18

1;2;1;2;2; - 19

Chunk Mapping Report

;1;2;3;6; - 5
0;1;2;3;6; - 0
;1;2;3;6; - 4
0;2;2;3;6; - 7
0;1;2;3;4; - 12
2;1;2;3;6;7; - 16

0;1;1;2; - 9
;1;1;2;3; - 2
0;1;1;2;4; - 8

1;3;;2;0;4; - 22
1;3;1;2;0;4; - 20
1;3;;2;0;4; - 21

1;3,40,5;1;2;2;2; - 1

4;66,1;15;;4; - 3

1;2,40,5;1;2;3; - 6

4;60,1;4;4; - 10

0,1;1;2;3;3;5; - 11

4;3;3;3;4; - 13

7;1;2,3;4;5; - 14

7,1;2;3;3;6; - 15

0;2;1;2;4; - 17

1;0;1;2;4; - 18

1;2;1;2;2; - 19

Chunk Mapping Report

2;;4;6;3; - 14
  2;4;4;6;3;2; - 11
    2;4;4;6,4;3;2; - 1
    2;4,1;4;6,1;3;2; - 10
    2;5;4;6,4;3; - 20
    2;4;4;6,4; - 21
  2;;4;6;3; - 13
    2;3;4;6;3; - 5
    2;1;4;6;3;;1; - 12
    2;3;3;6;3; - 19

  4;4,5;6,4;3; - 8
    4;4,5;6,4;3;2; - 0
    4;4,5;6,4;3;4;2; - 7

  7;;3,3;,5;6; - 18
    7;;3,3;4,5;6; - 3
    7;;3,3;5,5;6; - 17

    1;;3;3,4;5;7; - 2

    1;0;0,1,2;4;4; - 4

    1;1;0;1;2;3,3;4; - 6

    1,1;;1;4;4; - 9

    0;1;0,1,2;3,1; - 15

    2,1;1;1;4;4; - 16

Chunk Mapping Report

;1;2;3;6; - 5
0;1;2;3;6; - 0
1;1;2;3;6; - 4
0;2;2;3;6; - 7
0;1;2;3;4; - 12
2;1;2;3;6;7; - 16
1;1;2;3;7; - 24

0;1;1;2; - 9
0;1;1;2;3; - 2
0;1;1;2;4; - 8
0;1;1;2;5; - 23

1;3;;2;0;4; - 22
1;3;1;2;0;4; - 20
1;3;;2;0;4; - 21

1;3,40,5;1;2;2;2; - 1

4;66,1;15;;4; - 3

1;2,40,5;1;2;3; - 6

4;60,1;4;4; - 10

0,1;1;2;3;3;5; - 11

4;3;3;3;4; - 13

7;1;2,3;4;5; - 14

7,1;2;3;3;6; - 15

0;2;1;2;4; - 17

1;0;1;2;4; - 18

Chunk Mapping Report

1;2;1;2;2; - 19

Chunk Mapping Report

;1;2;3;6; - 5
0;1;2;3;6; - 0
1;1;2;3;6; - 4
0;2;2;3;6; - 7
0;1;2;3;4; - 12
2;1;2;3;6;7; - 16
1;1;2;3;7; - 24

0;1;1;2; - 9
0;1;1;2;3; - 2
0;1;1;2;4; - 8
0;1;1;2;5; - 23

1;3;;2;0;4; - 22
1;3;1;2;0;4; - 20
1;3;;2;0;4; - 21

1;3,40,5;1;2;2;2; - 1

4;66,1;15;;4; - 3

1;2,40,5;1;2;3; - 6

4;60,1;4;4; - 10

0,1;1;2;3;3;5; - 11

4;3;3;3;4; - 13

7;1;2,3;4;5; - 14

7,1;2;3;3;6; - 15

0;2;1;2;4; - 17

1;0;1;2;4; - 18

Chunk Mapping Report

1;2;1;2;2; - 19

Chunk Mapping Report

;1;2;3;6; - 5
0;1;2;3;6; - 0
;1;2;3;6; - 4
0;2;2;3;6; - 7
0;1;2;3;4; - 12
2;1;2;3;6;7; - 16

0;1;1;2; - 9
;1;1;2;3; - 2
0;1;1;2;4; - 8
0;1;1;2;5; - 23

1;3;;2;0;4; - 22
1;3;1;2;0;4; - 20
1;3;;2;0;4; - 21

1;3,40,5;1;2;2;2; - 1

4;66,1;15;;4; - 3

1;2,40,5;1;2;3; - 6

4;60,1;4;4; - 10

0,1;1;2;3;3;5; - 11

4;3;3;3;4; - 13

7;1;2,3;4;5; - 14

7,1;2;3;3;6; - 15

0;2;1;2;4; - 17

1;0;1;2;4; - 18

1;2;1;2;2; - 19

1;1;2;3;7; - 24

Chunk Mapping Report

;1;2;3;6; - 5
0;1;2;3;6; - 0
1;1;2;3;6; - 4
0;2;2;3;6; - 7
0;1;2;3;4; - 12
2;1;2;3;6;7; - 16

0;1;1;2; - 9
0;1;1;2;3; - 2
0;1;1;2;4; - 8

1;3;;2;0;4; - 22
1;3;1;2;0;4; - 20
1;3;;2;0;4; - 21

1;3,40,5;1;2;2;2; - 1

4;66,1;15;;4; - 3

1;2,40,5;1;2;3; - 6

4;60,1;4;4; - 10

0,1;1;2;3;3;5; - 11

4;3;3;3;4; - 13

7;1;2,3;4;5; - 14

7,1;2;3;3;6; - 15

0;2;1;2;4; - 17

1;0;1;2;4; - 18

1;2;1;2;2; - 19

;1;2;3;6; - 4
0;1;2;3;6; - 0
1;1;2;3;6; - 3
0;2;2;3;6; - 8
0;1;2;3;4; - 12
2;1;2;3;6;7; - 16

0;1;1;2; - 20
0;1;1;2;3; - 1
0;1;1;2;4; - 19

1;3;;2;0;4; - 22
1;3;;2;0;4; - 9
1;3;1;2;0;4; - 21

4;66,1;15;;4; - 2

1;2,40,5;1;2;3; - 5

1;0;1;2;4; - 6

1;3,40,5;1;2;2;2; - 7

1;2;1;2;2; - 10

4;60,1;4;4; - 11

4;3;3;3;4; - 13

7;1;2,3;4;5; - 14

7,1;2;3;3;6; - 15

0;2;1;2;4; - 17

0,1;1;2;3;3;5; - 18

Chunk Mapping Report

;1;2;3;6; - 7
1;1;2;3;6; - 2
0;1;2;3;6; - 6
2;1;2;3;6;7; - 8
0;2;2;3;6; - 10
0;1;2;3;4; - 13

1;3;;2;0;4; - 12
1;3;1;2;0;4; - 0
1;3;;2;0;4; - 11

0;1;1;2; - 22
0;1;1;2;3; - 1
0;1;1;2;4; - 21

1;2,40,5;1;2;3; - 3

1;0;1;2;4; - 4

4;60,1;4;4; - 5

1;3,40,5;1;2;2;2; - 9

4;3;3;3;4; - 14

7;1;2,3;4;5; - 15

1;2;1;2;2; - 16

0,1;1;2;3;3;5; - 17

4;66,1;15;;4; - 18

7,1;2;3;3;6; - 19

0;2;1;2;4; - 20

Chunk Mapping Report

2;;4;6;3; - 14
2;4;4;6;3;2; - 11
2;4;4;6,4;3;2; - 1
2;4,1;4;6,1;3;2; - 10
2;5;4;6,4;3; - 20
2;4;4;6,4; - 21
2;;4;6;3; - 13
2;3;4;6;3; - 5
2;1;4;6;3;;1; - 12
2;3;3;6;3; - 19

4;4,5;6,4;3; - 8
4;4,5;6,4;3;2; - 0
4;4,5;6,4;3;4;2; - 7

7;2;3,3;,5;6; - 18
7;2;3,3;4,5;6; - 3
7;2;3,3;5,5;6; - 17

1;2;3;3,4;5;7; - 2

1;0;0,1,2;4;4; - 4

1;1;0;1;2;3,3;4; - 6

1,1;2;1;4;4; - 9

0;1;0,1,2;3,1; - 15

2,1;1;1;4;4; - 16

Chunk Mapping Report

2;;4;6,4;3; - 3
2;4;4;6,4;3;2; - 0
2;5;4;6,4;3; - 2
2;3;4;6;3; - 10
2;4;4;6,4; - 15
2;4,1;4;6,1;3;2; - 18
2;1;4;6;3;;1; - 19

7;2;3,3;,5;6; - 7
7;2;3,3;5,5;6; - 5
7;2;3,3;4,5;6; - 6

4;4,5;6,4;3; - 17
4;4,5;6,4;3;2; - 8
4;4,5;6,4;3;4;2; - 16

2;3;3;6;3; - 1

1;2;3;3,4;5;7; - 4

2,1;1;1;4;4; - 9

1;1;0;1;2;3,3;4; - 11

1,1;2;1;4;4; - 12

0;1;0,1,2;3,1; - 13

1;0;0,1,2;4;4; - 14

Chunk Mapping Report

2;3;;6;3; - 3
2;3;3;6;3; - 1
2;3;4;6;3; - 2
2;1;4;6;3;;1; - 20

2;;4;6,4;3; - 5
2;4;4;6,4;3;2; - 0
2;5;4;6,4;3; - 4
2;4,1;4;6,1;3;2; - 8
2;4;4;6,4; - 14

4;4,5;6,4;3; - 17
4;4,5;6,4;3;2; - 7
4;4,5;6,4;3;4;2; - 16

7;2;3,3;,5;6; - 19
7;2;3,3;5,5;6; - 11
7;2;3,3;4,5;6; - 18

1;2;3;3,4;5;7; - 6

2,1;1;1;4;4; - 9

1;0;0,1,2;4;4; - 10

1;1;0;1;2;3,3;4; - 12

0;1;0,1,2;3,1; - 13

1,1;2;1;4;4; - 15

LLN Simulator
Technical Reference

Mark E. Gilbert
Undergraduate Honors Thesis
Lee Honors College
Western Michigan University
Kalamazoo, Michigan, USA

Table of Contents

I. Introduction

This is the Technical Reference for the LLN Simulator (LLNS), Version 1.0. This document describes how LLNS is implemented using Microsoft Access 2.0. The core of this document is broken up into four main components: learning a new memory, abstraction, querying (the three main pieces of LLN), and verification (where the implementation undergoes an analysis to confirm that the code written adequately simulates a LLN network implemented in silicon).

The two main tables of LLNS are "LLN Nodes" and "LLN Links". These two tables and their fields are described below:

LLN Nodes:

LLN Node Key	Counter (Long Integer with AutoIncrement)
Weight	Double (Float)
Category	Long Integer
Cell	Memo (Text - 64000 bytes max)
CatPosition	Long Integer

LLN Links:

LLN Link Key	Counter (Long Integer with AutoIncrement)
Chunk Key	Long Integer
Level	Long Integer
Subsumed By	Long Integer
LLN Node A	Long Integer
LLN Node B	Long Integer

Each node in the table is stored as a record in the LLN Nodes table. Each link in the table is stored as a record in the LLN Links table, and stores a key reference to each of the two nodes that it links using the LLN Node A and LLN Node B fields.

II. Learning a new memory

The routine that performs the new memory addition is called TrainMem, and is found in the TrainMem module. This routine will first parse the string (memory) passed to it, then set create new nodes/find existing nodes to store the pieces of the memory, then set up the links between those nodes, and then call the Abstract routine to complete the memories' integration into the network.

The parser must parse string expressions of the following type:

S :- B;
B :- a | a;B | a,B
a :- *r* | λ

where $r \in I$ and λ is the empty character (meaning that nothing is in its place, e.g., as is the case between the first and second semi-colon here, ";;"). Each terminal "a" becomes a node in the memory. The nodes that are separated by commas are called "units", while groups of cells separated from one another by semi-colons are called "cells". Each unit has two numbers which help to distinguish it from others: Category and CatPosition. The Category number of a unit corresponds to the cell to which it belongs in a particular memory (1..number of cells of memory), while CatPosition corresponds to the unit's position within the category (1..number of positions in a given category). The following is an example:

1,2,3,41;,5,67;;7,8,,19;;

Nodes 1-4 belong to cell 1, and therefore will have their Category positions set to 1. Nodes 5 & 6 belong to cell 2, there are no nodes in cell 3, and nodes 7-9 belong to cell 4. Within cell 1, there are four units: "1", "2", "3", and "41". These four will have their Category fields set to 1, while their CatPosition fields will be set according to their position within cell 1 (i.e., "1" will get 1 in the CatPosition field since it is first, "2" will get 2, etc.).

In cell 2, there are 3 units. The first unit is unknown (hence nothing is placed in the first unit's spot), while the second unit is "5" and the third is "67". Only units "5" and "67" will ultimately be converted to nodes, both receiving Category values of 2, and CatPosition values of 2 and 3, respectively.

Cell 3 is completely empty, so nothing will ultimately be done with this.

Cell 4 has three units in it: "7", "8", and "19". All three will have their Category fields set to 4, and will have their CatPosition fields set to 1, 2, and 4, respectively (please note that unit 3 in cell 4 is empty).

Cell 5 is also completely empty, so nothing will ultimately be done with this.

There are several additional things that are important to note: 1) that the entire string is ended with a semi-colon - this terminating character is required for any and all strings; 2) that the units can be multiple digits (i.e., "67"); 3) there are no negative units - the system will ignore (and therefore will not create nodes for) negative values for units; and 4) there are only integer units - the system will truncate all numbers to the right the decimal place.

The parsing begins by breaking the string up into a two-dimensional array of integers. The first dimension is the category that the units fall under, while the second dimension is the category position that the unit falls into. If there is a particular unit missing (i.e., is unknown), then a -1 is filled in. This will serve as a flag later to the program.

Once this 2-D array has been completed, the parsing code has completed. The system will now search for nodes already existing in the network that match (according to Category, CatPosition and Cell) the units just parsed. The units that are brand new to the database will have new nodes created for them. The system will track the list of nodes that the 2-D array of units are now represented by.

The system will now run a quick query on the database to find out what the last ChunkKey assigned was. The global variable ChunkKey (found in the Global Variables module) is set to this field, and then is automatically incremented. The new value for ChunkKey will become the Chunk Key of the new memory in the middle of being created.

The system takes the list of nodes just created/found, and sets up a new set of links between them. It sets Chunk Key to the incremented value found above, and Level to 0 (since this is a memory). It then increments Chunk Key.

Finally, the Abstract routine (described in the next section) is called to complete the integration of this memory into the system. The Abstract routine is passed two arguments: Chunk Key - 1 and 0 (the first is the chunk to be abstracted, and the second is the level that it is on). If this routine returns A_ERROR, an error message here is put out, and the TrainMem exits. If the Abstract routine returns A_NOERROR, then the TrainMem routine simply returns.

This routine is called by the TrainNow command button's OnClick event procedure once for each memory entered by the user in the NewMem text box. It is also called by the LoadAndTrain command button's OnClick event procedure. The latter event procedure takes the path and file name entered by the user in the MemoryFile text box, opens it, reads in the memories listed in it into an array of strings, and then calls TrainMem once for each string in the array. Once all string in this array have been trained in, the event procedure exits.

III. Abstraction

The routine that performs this operation is called "abstract", and is found in the Abstraction module. This function takes two arguments: the "Chunk Key" of the chunk to be abstracted (Chunk), and the level that that chunk is on (PrevLevel).

The routine begins by setting two invisible fields on the Train Form called "TempKey" and "TempLevel". These fields are used by the next queries to collect the proper information about the chunk to be abstracted. First, two delete queries are run that will flush out two tables: Abstraction T2 - A and Abstraction T2 - B. These two tables are intermediary steps in the creation of the Abstraction T2 table. The queries will append to them (described next) could not simply re-create them since they contain multiple counter fields, and the maketable queries cannot have multiple counter fields as output fields, but append queries can. The queries which append to these tables are: Abstraction Q2 GetANodes and Abstraction Q2 GetBNodes. These two queries work exactly like their Q1 counterparts, with two exceptions: 1) these two append to tables, while the counterparts were merely select queries; and 2) these two append queries use the invisible "TempKey" field on the Train Form instead of the parameters "[P1]" that is set by the function. These two look at the nodes that are set in the A slot and B slot, respectively, of the LLN Links table, and that belong to the Chunk passed in, and are run after the delete queries mentioned above do. Next, a maketable query called Abstraction Q2 MakeTable runs, and creates a table called Abstraction T2. This maketable query works exactly like the Abstraction Q1 select query, except the former creates a new table. It combines the output of the two append queries, and returns some other information about the nodes as well (Weight).

Next, the code needs to compile a list of chunks that use **any** of the nodes listed in the Abstraction T2 table. It first runs two maketable queries: Abstraction Q3 AHalf and Abstraction Q3 BHalf. These two queries will return the list of chunks that are joined to any "LLN Node A" field and any "LLN Node B" field, respectively, of any record in the LLN Links table. They will only return chunks that are on the same level as the chunk that is being abstracted. This is achieved by the queries checking the value of the "TempLevel" text boxes on the Train Form. This field was set earlier to the level passed into the function. These two maketables create the Abstraction T3 - A and Abstraction T3 - B tables, respectively. The next task is to combine these two listings together, and create a new table out of them. The query that performs this operation is the Abstraction Q3 MakeTable query, and the table it creates is the Abstraction T3 table. This query will not return Chunk (since this is the chunk that is going to be abstracted) - this is achieved by the queries checking the value of the "TempKey" text boxes on the Train Form. Once this table has been created, the code opens a recordset on it (IndexMems).

Next, the code begins a While loop that will look at each of the chunks pointed to in IndexMems. It will first get a list of the nodes that Chunk and the chunk pointed to by IndexMems share, calculating the closeness ratio for each pair. This is done by the Abstraction Q4 set of queries. First, a pair of queries called Abstraction Q4 GetANodes and Abstraction Q4 GetBNodes will return all of the nodes that Chunk or the chunk pointed to by IndexMems have. Then, the two queries are combined into the Abstraction Q4 AllNodes, which combines these two recordsets into one for easier management. Also, the Weight value for these nodes is returned to aid in the closeness ratio calculation a little later in the code. Next, a Find Duplicates... query called Abstraction Q4 SharedNodes is run. This query is Grouped so that the duplicated nodes

(the ones that the two chunks share) are listed only once in the recordset. This query is used in by the Abstraction Q4 Ratio Numerator query, which sums up the returned Weights. As the name implies, this sum of the weights becomes the numerator for the ratio calculation: the sum of the weights of the nodes that are shared between the two chunks. A check is done on the IndexMems pointer. If there are no records returned, it means that there were no other chunks that shared any nodes with Chunk. If this is the case, the code exits - abstraction is complete for this Chunk. If there are chunks returned in the table, the next While loop handles them.

In the main While loop, three recordsets are opened: IndexNumer (the SQL of which is the same as the Abstraction Q4 Ratio Numerator query) and IndexDenom1 and IndexDenom2 on the Abstraction Q4 DenomNodes Diff query. The former of these recordsets is a lot easier to understand - it returns the sum of weights for the nodes that are shared between the two chunks being examined. The second is easy to understand what it returns, but needs a little bit of detail as to how it achieves it.

The Abstraction Q4 DenomDiff recordset is opened using the Chunk variable and the current chunk pointed to by IndexMems. This recordset returns the list of nodes that are **not** shared by the two chunks passed to it. This recordset pulls information from the Abstraction Q4 DenomDiff Sub query. The Abstraction Q4 DenomDiff query checks the Sub query's "1" and "2" fields for null: if both are null, the record to which they belong is not returned, the Weight and LLN Node Key fields are returned, and the contents of the recordset are sorted in ascending order first by Category and then by CatPosition.

The Abstraction Q4 DenomDiff Sub query links the LLN Nodes table in directed joins to the Abstraction Q4 DenomNodes 1 query's LLN Node Key and the Abstraction Q4 DenomNodes 2 query's LLN Node Key. The output fields are the LLN Node Key, Weight, the LLN Node Key fields from both of the subqueries, the Category and the CatPosition. The subquery's LLN Node Key fields are checked for Null: if either is null - and consequently if both are - the record to which they belong is returned. Note the result of this and the above query - only those records where one or the other LLN Node Key field from a subquery is ultimately kept.

The Abstraction Q4 DenomNodes 1 subquery returns the list of nodes that belong to the chunk [P1] (the first parameter set by the code). This is achieved by linking the LLN Nodes table in a directed join on the LLN Node Key to the Abstraction Q4 DenomANodes 1 and the Abstraction Q4 DenomBNodes 1 queries'' LLN Node Key fields. The LLN Node Keys from these two subqueries are checked for null: if either are not null, the record to which they belong is kept in the output. The Abstraction Q4 DenomNodes 2 query works the same way, except it joins the LLN Nodes table to the Abstraction Q4 DenomANodes 2 and the Abstraction Q4 DenomBNodes 2 queries, and prompts for parameter [P2] (the second parameter set by the code. The "A" versions of the above subqueries return the nodes linked to any LLN Node A field of any link in the LLN Links table, while the "B" versions look at the LLN Node B fields.

Back to the code. There are two pointers that are opened on the Abstraction Q4 DenomDiff recordset - IndexDenom1 and IndexDenom2. The system checks IndexDenom1 for BOF - if it is true, the two chunks being examined are identical. The CurRatio is set to 1, and the two recordsets are closed. If not, the code performs a quick check for a rare, but possible scenario: the two chunks being examined are identical except that one has a node that the other does not. In this case, there will be only 1 record record by the IndexDenom1 and IndexDenom2 recordsets. The test for this condition is easy: if IndexDenom1 has reached the EOF after only a

MoveFirst and a MoveNext call (performed prior to this), then this condition is true. In this case, the DenomDiff variable should get half of the weight that is pointed to by IndexDenom2 (consistent with the rest of the code).

Next, the calculation of the weight contribution from the non-shared nodes is computed. This contribution is stored in DenomDiff, and will be used later when the closeness ratio is calculated. The While loop that is executed performs the following: if there are two units from the same Category and CatPosition, then average the weights of these two units and add this average to DenomDiff. If there is only one unit in the list for a given Category and CatPosition combination (meaning that the corresponding cell in the other chunk is missing, i.e., unknown), average the unit that is there with 0 and add this average to DenomDiff. Perform these averages and additions until the entire list pointed to by IndexDenom1 and IndexDenom2 has been exhausted.

Once DenomDiff has been calculated, the ratio calculation is carried out. A series of checks is made to see what should be done with the current chunk based on the ratio:

If the ratio is equal to 1, then the two chunks are identical
    Remove the newer of the two from the tables
Else, if the ratio is greater than the vigilance setting then:
    If the ratio is greater than MaxRatio, then the current chunk becomes the new MaxCurMem; reset MaxRatio, and throw out the old chunk (done by setting RemoveMem to the value of the old chunk)
    Else, if the ratio is equal to MaxRatio, we have the situation where there are two chunks that are equally close to Chunk. In this case, throw out the chunk with the higher of two Chunk Keys. This chunk being thrown out is the younger of the two chunks.
    Else, if the ratio is less than MaxRatio, throw out the new chunk
Else, if the ratio was not higher than the vigilance, throw the new chunk out

The If clause above uses the "qdelChunk - TrainMem" delete query to remove the younger of the two chunks from the system. If this were not done, input vectors that had already been trained in would be retrained in as new memories, and would appear more than in the tables.

From here, RemoveMem holds the chunk that is to be removed from the list, so the system moves on to the next chunk in the table, and the chunk to be removed is removed at this point. When this loop has completed, there will be at most a single chunk that Chunk will be abstracted with, either by altering the current chunk's subsuming concept (if such a subsuming chunk exists), or by creating a new chunk to subsume both the current and the new chunks. The code will perform a quick check to see if any of the chunks examined had a closeness ratio to Chunk greater than Vig, the vigilance setting. It does this by checking the current value of CurMaxMem. CurMaxMem was initialized to -1 before the loop began. If it has kept this value throughout, it means that there was not a single chunk that had a ratio greater than Vig. If this is the case, then abstraction has completed at this level, so simply return.

If the code makes it passed the above check, the system will then compile a list of the current chunk's links (done by the Abstraction Q5 GetLinks select query) and see if there is a subsuming concept for it. If there is, it then runs another query (Abstraction Q4 SharedNodes)

which will return the nodes that the current chunk and the new chunk share. This set of nodes will have their weights adjusted to reflect a new chunk being subsumed, and the new chunk will have the Subsumed By fields of its links set to the Chunk Key of the subsuming chunk. If there is not a subsuming node, a query is run to see which nodes are shared by the current chunk and the new chunk. This list of shared nodes becomes formed into a new chunk, their weights being altered as in the previous scenario, and both the current and the new chunks have their Subsumed By fields set to the newly created chunk's Chunk Key. The Chunk Key of the chunk that was either altered or created is saved in a temporary variable (NextChunk). This variable represents the chunk that is to be looked at by the Abstract function in the next call (i.e., will become Chunk in the next call). Before this call is made, however, there is one more recordkeeping function the needs to be performed - that of setting the "Subsumed By" fields of the newly subsumed chunk(s).

The code first checks the value of IndexLinks![Subsumed By] field. This recordset pointer currently points to the list of links that belong to CurMaxMem. If this field is Null or less than zero, then it is not already subsumed by a chunk, and as a result needs its fields set to the newly created chunk. This is done via a While loop. The loop will walk through the links of CurMaxMem, and set their "Subsumed By" fields to the value of NextChunk. Once this loop ends, the IndexLinks pointer is closed and is reopened on Chunk. This chunk's links have their "Subsumed By" fields also set to NextChunk in the same type of While loop.

Once this second loop ends, the recordset pointers IndexLinks, IndexNumer, IndexDenom, and IndexShared are all closed, and the recursive call to Abstract is made using "NextChunk" for Chunk and "PrevLevel + 1" for PrevLevel. The return value of this recursive call is saved in Ret, and Ret is returned as the return value of the current call to Abstract.

IV. Querying

There are two methods of querying the network: search for close chunks, and the chunk mapping report. The former tries to find all chunks that contain all of the attributes that the user presents to it, while the latter will map out all chunks in the system according to the hierarchical relationships established during the abstraction process. This section will describe both in the above order.

The chunk search begins by parsing the attributes presented to it by the user (the user types this list into the Single Chunk text box. This list of attributes must follow the same guidelines as described in Section 1 Learning a New Memory above. As a result, the first part of the code for the SearchForChunk OnClick event procedure will parse this list, and establish a temporary table with the corresponding LLN Node Keys in it. The parsing code is identical to the code found on the Train Form (please refer to Section 1 above).

Once this table, called Query T1, is complete, a maketable query (called Query Q1) is run to compile a list of links that specify (in either the A or B slot of the link) one or more of the nodes in the Query T1 table. It accomplishes this by linking the LLN Links table to two copies of the Query T1 table - one copy to LLN Node A, and the other to LLN Node B. The joins are directed from LLN Links to the other tables; the output is the LLN Link Key, the Chunk Key (which is sorted ascending), the Level, Subsumed By, LLN Node A (renamed as A), and LLN Node B (renamed as B). The last two of these fields has their criterion rows set so that if both are null, the record is not returned. If both are null, it means that the current link does not connect to any of the nodes in the Query T1 table, and therefore should not be returned. Query Q1 makes the Query T2 table.

Section Three of the code is where the chunks represented in the Query T2 get weeded out. A chunk will not remain in the table if its links (the ones returned) does not link in some way all of the nodes listed in the Query T1 table. This weeding out is accomplished by setting two pointers, IndexA and IndexB, to point to the contents of the two tables, Query T1 and Query T2, respectively. IndexB is set to point to the first chunk in the Query T2 (all links for a given chunk are grouped together in the table due to the Query Q1's sorting the Chunk Key ascendingly). Then, IndexA will walk through the nodes in the Query T1 table. It looks for the current node in the chunk pointed to by IndexB. It checks both the A and the B slots of the links of that chunk. If the node is not in any of the links, that chunk is marked for removal by setting the RemoveMem variable to its Chunk Key. Also, if the chunk is to be removed, the IndexA is forwarded to EOF (so that the loop will exit immediately). If the loop completes and RemoveMem was never set to something non-negative, it means that the current chunk pointed to by IndexB contained all of the nodes in the Query T1 table, and therefore should remain in the Query T2 table.

After the loop completes, the value of RemoveMem is examined. If it is negative, nothing is done. If it is non-negative, the next while loop will remove from the Query T2 table the inks belonging to the chunk with RemoveMem as the value for Chunk Key.

Once this check is done, the code moves onto the next chunk in the table. This is accomplished by tracking the Chunk Key value of the chunk just examined in PrevMem. The IndexB pointer is forwarded to the first set of links with Chunk Key's having a value greater than PrevMem. This works whether or not the previous chunk was removed.

Once Section Three's code finishes, the nodes left in the table constitute the close chunks to the list of attributes that the user provided. Now, the object is to reverse parse the list of nodes back into the form that they were originally entered into (the form described in Section 1), or close to it. Before this begins, a check is made on the state of the Query T2 table. If it is empty, the code says so, and exits.

If the table is not empty, the Query T3 table is cleared, and a fresh pointer, IndexQ, is set to point to it. This table will hold the chunks in one record, and will contain the following information about them: Chunk Key, Subsumed By, Level, and the chunk itself (in a single string with the UnitSeparators and CellSeparators re-inserted). First, the nodes that are linked by the links in the Query T2 table are dumped into the Query Tmp table, one chunk at a time. Just before the code enters another node, it checks to see if that node is already present in the table. If it is, it moves on to the next one; else it enters it.

When this code finishes, a query is opened on the Query Tmp table that links in the Category, CatPosition, and Cell of each node (from the LLN Nodes table). Then, the next while loop rebuilds the original form of the chunk (minus any trailing blank units in a cell, or any trailing blank cells). It does this by looking at the value of the Category and CatPosition of the current node to be entered. It then adds in CellSeparators (into a temporary string variable called Tmp) until the current cell is reached, and then adds in UnitSeparators (into Tmp) until the current unit is reached. Then, it adds in the Cell of the node. Then, it moves onto the next node, and repeats the process.

When the last node has been entered, the while loop exits, and a final CellSeparator is appended to Tmp. Then, a new record in the Query T3 is created (using IndexQ), and the chunk is entered into the table. At this point, IndexB points to the last link record of the chunk just rebuilt. It is forwarded one record, and a EOF check is done on it. If EOF is true, CurMem is set to -1 to cause the outermost while loop to terminate. If not, CurMem is set to the Chunk Key value of the chunk now pointed to by IndexB.

This process is repeated for every chunk left in the Query T2 table. Once it completes, a report called the Query Network Report is opened on the Query T3 table, and this report will display the information stored there. Once this report is opened, the subroutine exits.

The second query method is the Chunk Mapping Report. This code begins by searching to see if there are any chunks to map. If not, the code says so, and then exits.

If there are chunks to map, the code clears out the Query T3 table, and then finds the highest level of chunk currently stored in the table. The code then compiles a list of chunks that are not subsumed by any other node, i.e., if their Subsumed By fields are set to -1, and are at this highest level. A recordset called IndexTable is opened on the Query T3 table, and then a While loop begins.

While the current level is greater than or equal to zero, the descendents of the current chunk are found using the CreateChunkMap function (see below), and added to the table. Then, a blank record is placed in the table using IndexTable, and the next chunk that is at the same level and is subsumed by no other chunk is looked at, and the process repeats. If there are no other chunks at this level that don't have other chunks subsuming them, the next lower level is looked at. Once this level gets to -1, the outer while loop exits.

Once this outer while loop exits, the Chunk Mapping Report is opened on the Query T3 table.

The CreateChunkMap function works by indenting subsumed chunks under the chunk subsuming them. For example:

```
aaa - i
  bbb - j
    ccc - k
    ddd - l
    eee - m
  fff - n

ggg - o
  hhh - p
  iii - q
```

shows that chunk aaa subsumes directly chunks bbb and fff, while bbb directly subsumes chunks ccc, ddd, and eee. There is a space placed between the chunk aaa and its descendents and chunk ggg and its descendents because these are unrelated chunk "lines".

The number that follows each chunk is the Chunk Key field, as returned by the report's underlying query.

The function takes three arguments: ML (maximum level), Level (the current level of the chunk being looked at), and Chunk (the chunk being looked at). This function is recursive: it looks for the descendents of Chunk, and then recurses to find the descendents of those descendents, etc..

The function begins by opening a recordset of its own on the Query T3 table called IndexQ. Then, the code gets a list of the nodes that Chunk links together, and dumps the reverse parsed chunk into the table. To get the proper indentation, the difference between ML and Level is calculated, and that number times 2 equals the number of indentation spacing required for the current chunk. So, for instance, if Chunk was on level 2, and ML was 5, there would be (5-2)*2, or 6 indentation spaces used for Chunk. The reverse parsed chunk is appended to this spacing, and the concatted string is added to the table.

Next, the Chunk's Level is tested. If Level is 0, then this was a memory, and therefore, does not have any descendents, so return. Else, continue.

Then, the recursive call to CreateChunkMap is made to find and dump the descendents of Chunk's descendents to the Query T3 table. When this completes, the code returns.

V. Verification

This section shows hows the algorithms used to implement LLNS are equivalent to the ones described in the design specifications for LLN (Gilbert 1997). The general method that will be employed to do this will be to look at the results of each algorithm: do both methods accomplish the same things, and do both prevent the same things from happening? In this section, the design specification of LLN will be called LLN, while the simulator version will be referred to as LLNS.

This section is broken up into three main sections: New memory addition, abstraction, and tree traversal.

New Memory Addition

The sensory apparatus mentioned in the design specifications is not part of LLN, although any type of sensory apparatus output that could be accepted by LLN can be accepted by LLNS (binary, integer, reals, imaginary, etc.) since LLNS stores the input as a string. In other words, as long as the input can be represented in English, it can be represented in LLNS.

Creation of a new category in LLN is explicit in the creation of a node for a particular unit in LLNS. However, the creation in LLNS is much simpler than in LLN. In LLN, the idea is for one class of nodes to respond favorably to the new unit, for example firing (in the biological sense) rapidly. This feature is cut out of LLNS, but the end result is the same: the node is put in its correct category.

The problem with having a pre-existing node becoming corrupted or damaged beyond usefulness is preserved in LLNS, but in a simpler way. In LLN, the idea was that an attribute was stored in a small neural network that was vulnerable to having damaging weight changes. In LLNS, the node's Cell field value could get changed, or the entire node could be removed from the system.

The creation of a new node or link wasn't well defined in LLN, and this was intentional - it really didn't matter to LLN *how* the nodes and links got created. As a result, the programmer has a lot of flexibility to do it how they see fit.

Abstraction

The search for the nodes used by the chunk to be abstracted in LLN was intended to be something like the following: through each of the attributes of the chunk out to all of the nodes, and see which ones fire for which attributes. In LLNS, this is done using MS Access' search routine, which returns all possible matches of the attribute to the node. It is more direct in LLNS, but it accomplishes the same thing.

The traversal of the nodes in LLN is done via following the links connecting to each of those nodes. In LLNS, this traversal is done, but it is done at the same time for all pairs of connected nodes. In other words, the implicit idea in LLN is that you start at one of the nodes, and follow its links until you end up back at that node, then you move on to the next node in the set. In LLNS, this is done for all nodes in the same step (this is done by the Abstraction Q3 MakeTable, and the table that it creates is Abstraction T3).

The weeding out of the memories in the LLNS table follows pretty close to LLN. The selection of the abstraction technique (concept creation or concept alteration) is the same in both systems, as is the carrying out of that technique.

Tree Traversal

The first method of querying - searching for a chunk in the network - is simpler in LLNS than in LLN. In both systems, the nodes representing the target chunk are found pretty much the same way, but the parting of the ways is in the search for links. In LLNS, the LLN Links table is accessed directly, and any link that connects two of the target nodes is returned. Then, this list if processed for each chunk. If a particular chunk contains all of the nodes (i.e., if there are links to all of the nodes) of the target chunk, that chunk remains in the list. If there is even 1 node missing from the list for a particular chunk, that chunk is removed from the table.

In LLN, a similar procedure is performed, but it requires that the links and nodes are visited one at a time, in separate steps.

The second method of querying - creating the chunk map - is not described in LLN since, as the design specifications put it: "There does not seem to be a need for a function of this sort if the network was being used by an inference engine or other AI system." (Gilbert 1997) In other words, this isn't a normal network operation, but is useful for seeing what the network has learned. It is a diagnostic utility of sorts - not designed to be used in normal operation.

Bibliography

Gilbert, Mark E. Link/Loop/Node Networks. Undergraduate research paper, Lee Honors College, Western Michigan University, Kalamazoo, MI, USA, 1997.