//==-- Outline

0: Theory and explanation

1: Storing a construct

2: Retrieving a construct

3: Charging constructs

4 Construct hiearchy

5: Working with upper tier constructs

6: Types of upper tier constructs

7: Data dimensions and construct relationships

8: Chronological constructs

9: MSC construct chain linking

10: Chrono linking

11: Types of complex assemblies

12: Goal seekers

13:

//==-- Table Of Contents

The goal is to create a model that allows us to store patterns in a scalable, algorithmic, and versatile way.

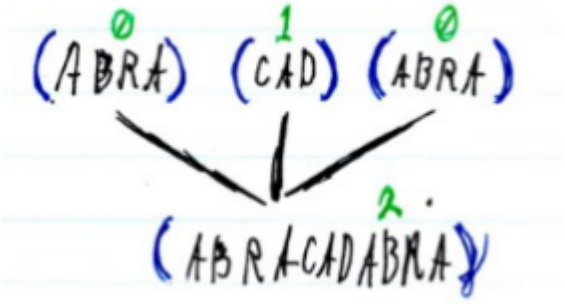
First we must break the data we want to work with down into a manageable format.

For out data to work with we will be using the string “ABRACADABRA”.

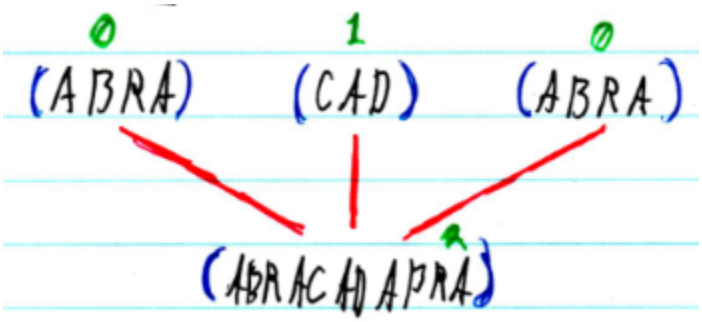
Looking at this string we can see several patterns that repeat, or are very similar. We must now find a way to extract these patterns.

ABRA|CAD|ABRA

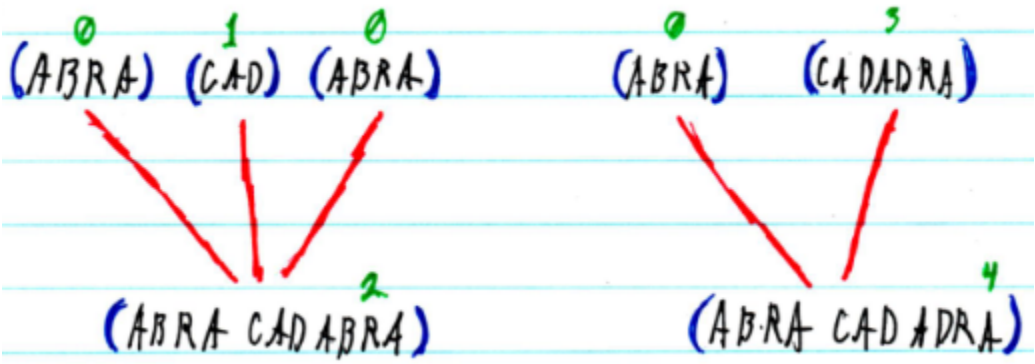
We can see from this example that the data is composed of two “ABRA” patterns sandwiching a “CAD” pattern between them.



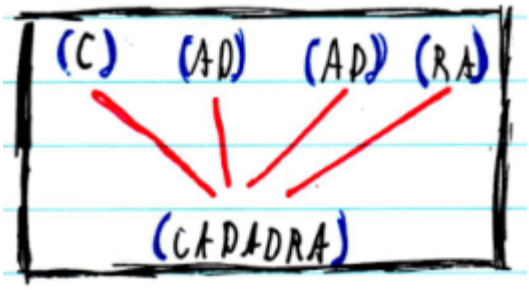
We have labeled the different patterns here so that you can easily identify them and see how they fit together to form the larger pattern.



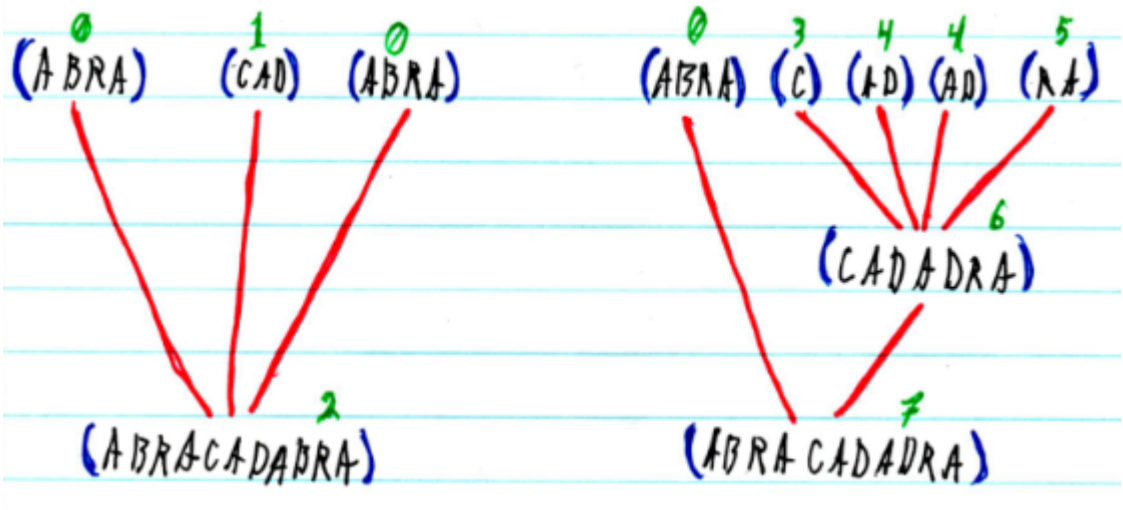
Now what happens if we create the same kind of representation with a slightly corrupted version of our original string next to our original?



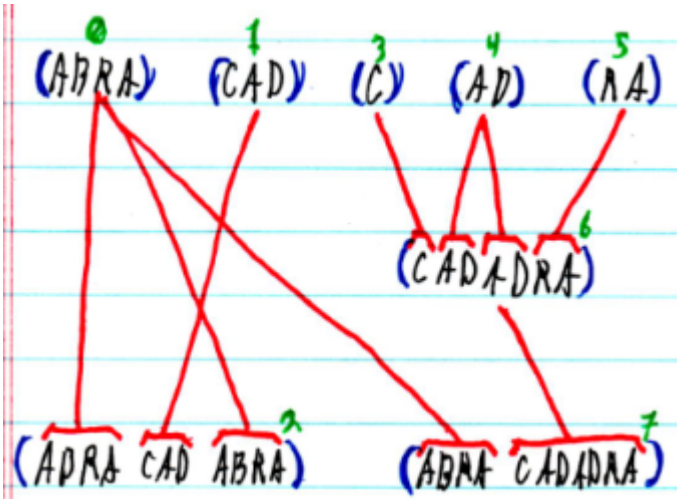
If we look closer “CADADRA” contains repeating patterns within itself that can then be broken down into smaller component patterns.



Now we can work that into our main example and we get the following result.



We can take any repeat patterns and only show them once, but link them in such a way that the more complex patterns share these component patterns in a coherent way. This way the components are only represented once.



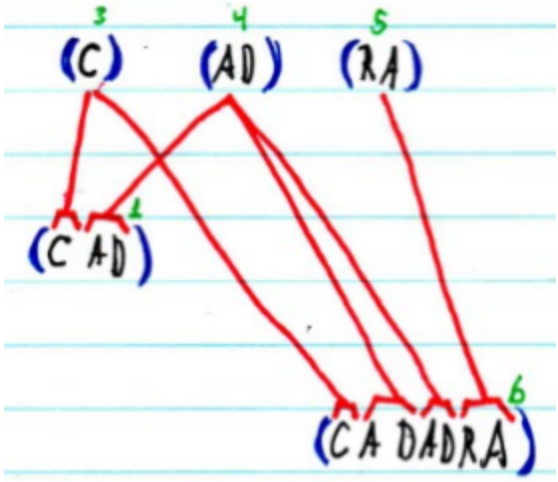
This collection of complex patterns and their component patterns we will henceforth refer to as networks.

Now looking closer at this we can see that shits pretty fucked. We have the (CADADRA) complex pattern that contains the (CAD) pattern, however, in the network the (CADADRA) is built out of (C), (AD), and (RA). Now if we are trying to only represent each component pattern once this is in conflict with our goals. Looking closer you will find more conflicts than the (CADADRA) pattern conflict.

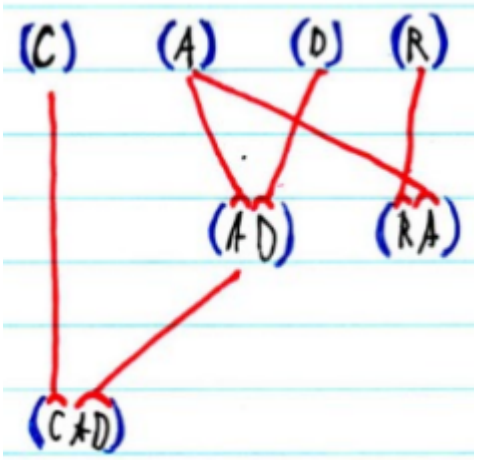
To solve this you could manually figure out every conflict and resolve them by hand, but this would be cumbersome and not algorithmically which is the goal we seek. The solution is then to come up with a method of consistently breaking these complex patterns into all of the component patterns in such a way that conflicts are easy and effectively found and eliminated.

To continue we will be using the shorter pattern (CADARDRA) during our attempt at finding a solution to this problem.

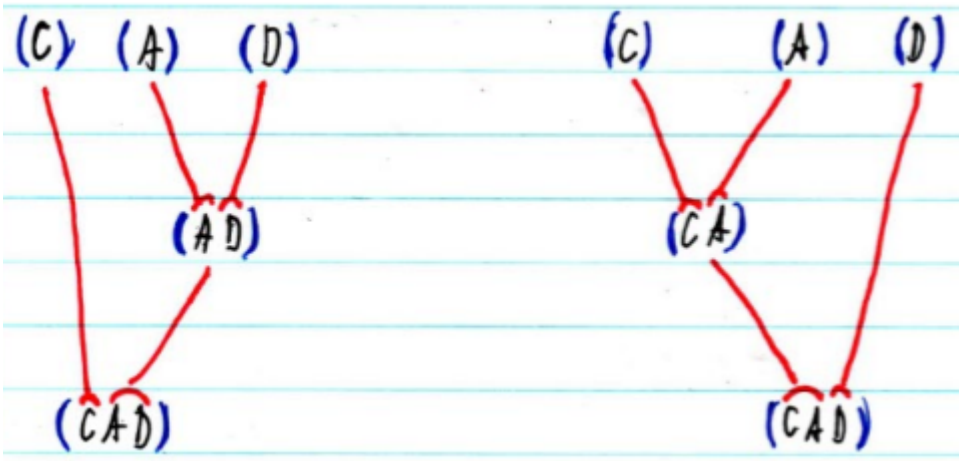
First let us take the (CAD) pattern and work it into the (CADADRA) pattern network.



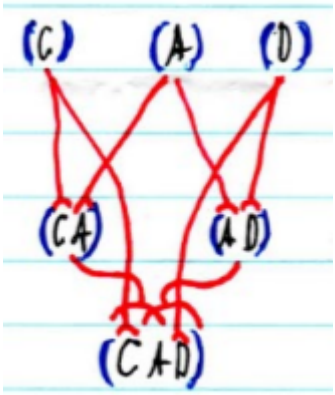
AH FUCK, the (CADADRA) patterns contains both the (CAD) and the repeat (AD) pattern! Well before we get all worked up it is worth noting that here we can see a pattern can be as small as a single letter in the case of (C). Although it would not really be a pattern at that point we will ignore that for now in the interest of simplicity. Looking at it with this realization we can then see that both (AD) and (RA) contain the (A) pattern. Well then, let us work that into our network here.



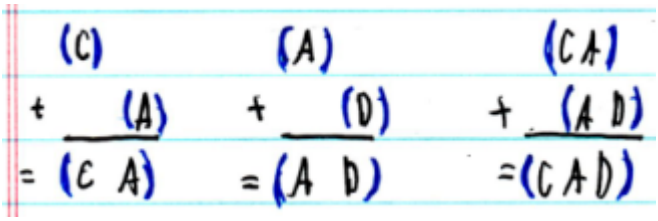
Now looking at the (CAD) pattern it could be said that it contains either (CA) (D) or (C) (AD). Both work in a network, but not both at once without creating two networks.



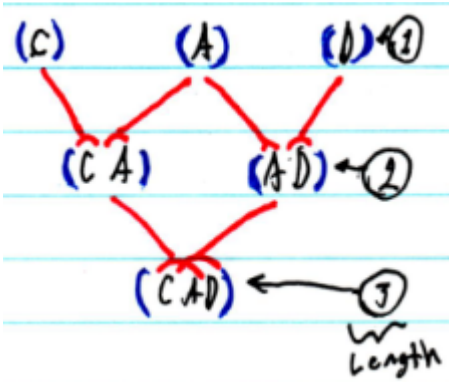
If we stick to the rule of no repeat patterns represented in a single instance it becomes obvious this will not work.



If we are treating (C), (A), and (D) as patterns then that would mean that (C) is the left portion of the (CA) pattern with (A) being the right portion of the (CA) pattern. So then could this logic be extended to the (CA) being the left portion of (CAD) with (AD) being the right portion of the (CAD) pattern?



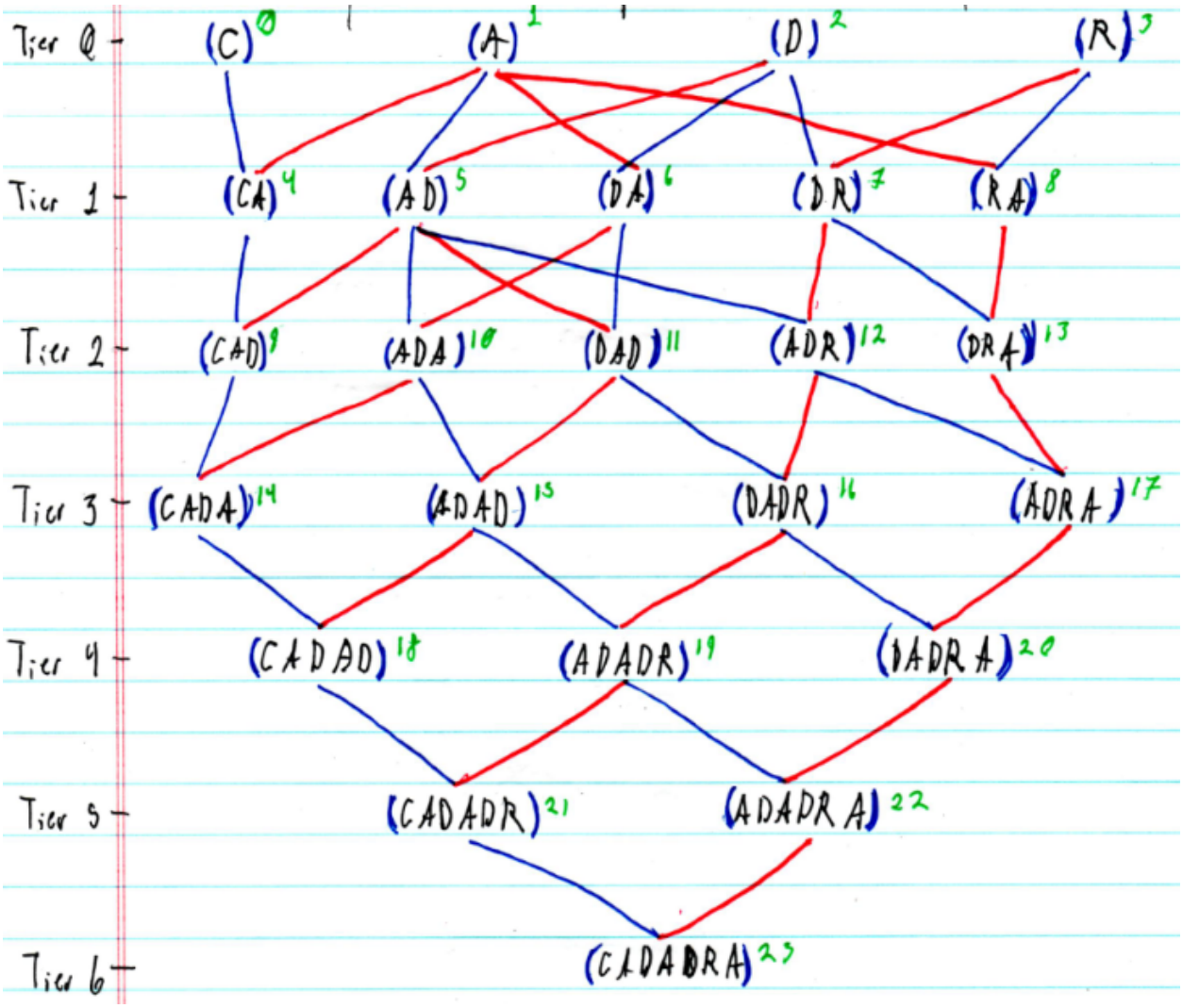
The result of this representation would create a network looking like this:



Using this model we can then build both (CADADRA) and (CAD) with no conflicts. Now, if from here on out we use a strict rule of using sub patterns to form the left and right portions of more complex patterns we can color code these connections. Blue will always be the color of the connection between the left portion and the pattern, and red will represent the right connection.

Looking at the previous figure there is another thing worth noting, and that is that each tier in the network is related to how long the patterns on that tier are. Notice that each tiers patterns have a length equal to one more than the tier below it, and one less than the above it. Not only that, but if we count tier 1 as the tier containing the patterns with a length of 1 then each tiers number is equal to the pattern length of its constituent patterns.

Using the new method of storing patterns let us go back to the (CADADRA) example and see what the new network looks like.



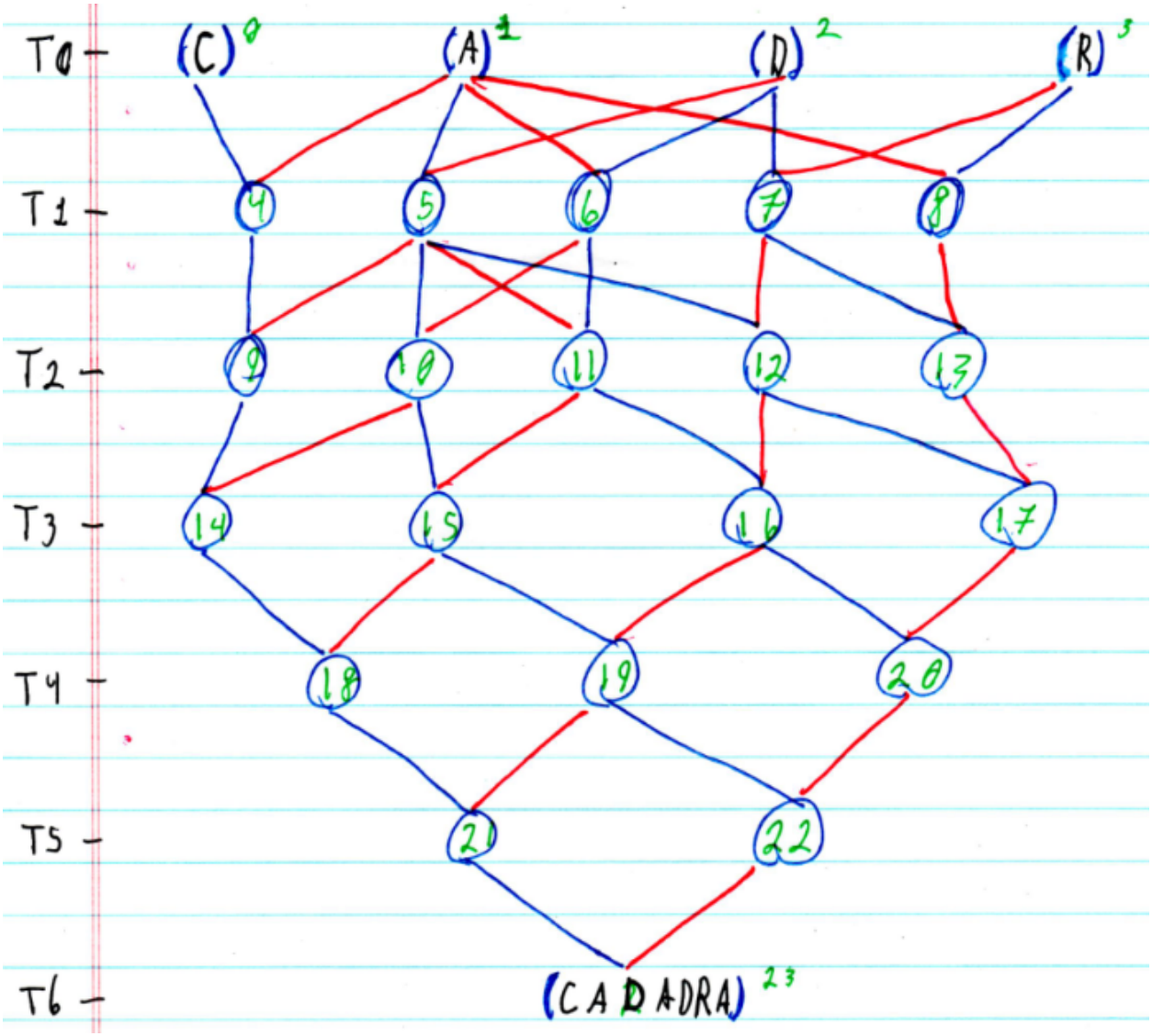
At first this seems to contain an excessive number of pattern chunks, which we will call nodes, however, it allows us to store the (CADADRA), (CAD), and (AD) patterns with no conflicts. What is more is that this allows us to store ANY pattern with the base components of (C), (A), (D), or (R) with no conflicts! The result of this attempt to resolve the conflict has resulted in a scalable model with a predictable structure.

There are some steps we can take to simplify things a bit at this time. For example we don't really need to write out each and every single pattern in the network. What I mean by this is that since each pattern is made up of lower patterns we can figure out the current pattern from the lower ones. This is possible due to the left and right patterns that make up the current pattern, by figuring them out and putting them together we get the upper tier pattern.

The only exception to this is when we get all the way to the lowest tier; those patterns must be representative of a unit of data, in our case a letter. Doing this allow a great simplification in the plotting of the data, and a huge decrease in the amount of memory needed to store the network should we store it in a computer. We can refer to this simplified pattern representation as a node. So a network is

composed of tiers of nodes which represent patterns. These patterns are not stored in the nodes themselves but in the order of connections to lower nodes.

Now, we need a way to reference the nodes in these networks now that we are not storing patterns in their entirety. We can do this by simply assigning an ID to each node. Now instead of having to know that what pattern each node represents to reference it we can just reference its node ID (NID). The next section will deal with extracting these patterns from the nodes, for now just know we can do it.



Before we continue we will look at what we have gathered so far about our model and some observations about its composition and behaviour.

1: In each network nodes representing the same pattern only appear once.

2: Networks are arranged in a tiered hierarchy.

3: Each tier in this network structure holds patterns that get increasingly complex.

4: The complexity of these patterns grows by one unit of the pattern for each tier climbed.

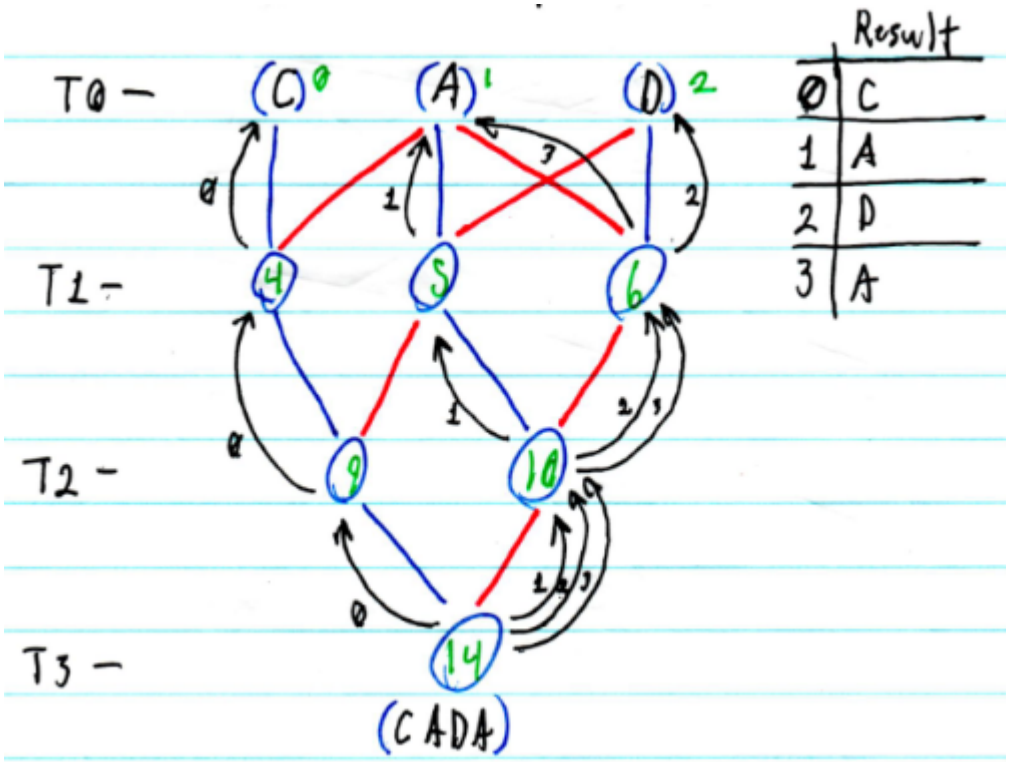
5: Each pattern in a tier is composed of two patterns in the tier below it, both a left and right portion.

6: Only two lower patterns are used to create a pattern for now, more on this later.

7: Although each pattern is composed of exactly two lower tier patterns, they can themselves make up any number of higher tier patterns.

8: Each node is represented by an ID rather than a pattern.

Now that we have that settled we will work on extracting a pattern from a node. For this let us use the example of the node #14 from our previous network. This node represents the pattern (CADA). A simple enough example I believe.



From (14) we can see that following left legs leads to the state node containing (C). Now we know that our patter is (CADA) so we need an (A). We can get this by following the right leg of (14) to (10) before resuming taking all left legs through (5) to the state node (A). So now that we need a (D) for our third letter we take the right path down two nodes and then the left as before. This results in (14) -> (10) -> (6) -> (D). Now for the last letter let us take the right legs down one more node than the last run before going left. (14) -> (10) -> (6) -> (A). Well we ran out nodes before we could continue on the left legs.

Now something you may have noticed about this is that for each bit of data needed we went one more leg down the right side than the previous run. This works out to be equal to the index of the data wanted (cou_Index) down the right legs before going down the left legs. Another relationship found here is that the number of data units recovered is equal to the height of the node in the tiered structure. So then data length is equal to node tier height plus one. So let's put this into some pseudo code.

0: Use #Right_Legs to track how many right legs to go down before going left.

1: Starting at current_Node go #Right_Legs down the right legs

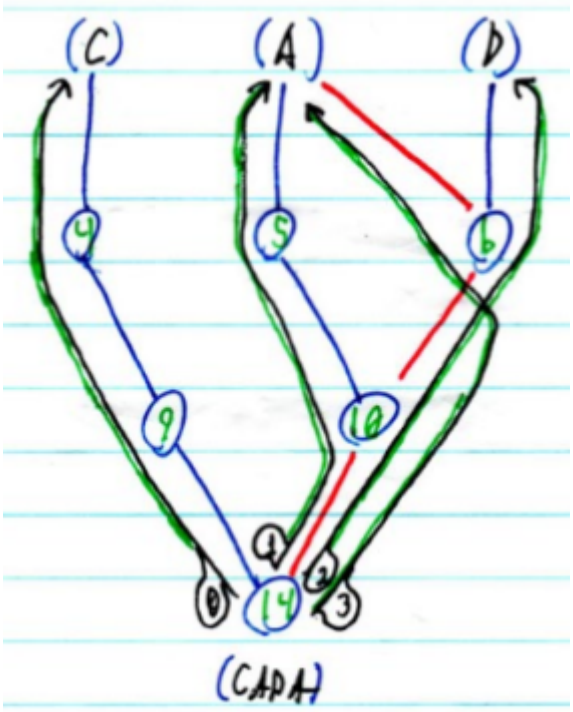
2: If the current tier is greater than 0 then go down left legs until tier 0 is reached.

3: Gather the data unit from the current state node on tier 0 into the Data array.

4: Increase #Right_Legs by one.

5: Repeat steps 1 – 5 until the #Right_Legs is equal to the current tier.

If we draw the network with just the paths taken to better show the output paths we get the following diagram.



We can now store and retrieve patterns by hand. As nifty as this is the ability to store and retrieve patterns by hand is not terribly useful in a practical sense. The next step is to find out how to store these networks algorithmically so that the building process can be automated by a machine to store any pattern we want into a network.

Now we begin by defining the structures we will be working with. After defining the structures we will then define the methods for working with the structures.

To start with we will define the nodes and what they consist of. There are two types of basic nodes that we are aware of at this point, the normal node, and the state node. Let us list all of the things known about these two types of nodes.

-Only the bottom nodes need to store a unit of data, these are the state nodes.

-Normal nodes have a set number of lower connections (Dendrites or Legs)

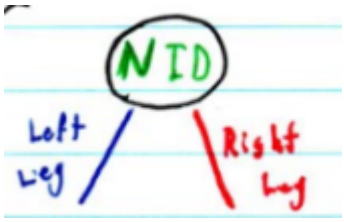
-State nodes do not have lower legs, only a state.

-Unlike legs there is no limit on the number of upper tier nodes that can link to a node. (Axons)

-In the nodes with legs the lower legs are strictly left and right, and can't be interchanged.

-Every node has an identifier unique to it used to reference the node with (NID).

From these we can draw an example of both types of nodes. First the normal node followed by the state node.



From here we can define the components of these two types of nodes. We have a NID and the lower legs along with the upper connections and their count for the normal nodes. For the state nodes we only need the state data unit and the upper connections and their count.

--Normal Node:

-NID

-Left Leg

-Right Leg

-#Upper_Tier_Left_Legs

-Upper_Tier_Left_Legs[Array]

-#Upper_Tier_Right_Legs

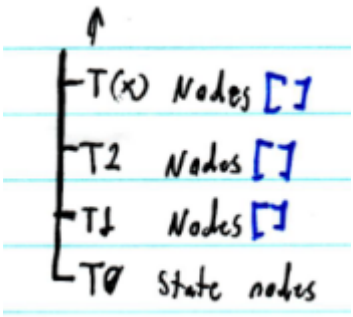
-Upper_Tier_Right_Legs[Array]

--State node:

-NID

-State

Now that we have the nodes defined we will move onto constructing a definition of the network. The networks are composed of nodes; these nodes are arranged in tiers with the lowest tier being the state nodes. Each tier in the network is an array of nodes. Here is a visual example where [] means it is an array.

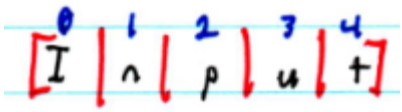


--Network:

-Tiers of nodes

-Number of tiers

That sums up the nodes and the network, but not only do those need defined we have to work with the input data as well. For this we will define an input pattern which is composed of states. Here is a visual example followed by the definition.



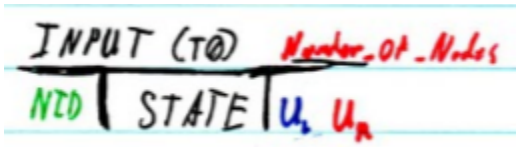
--Input Pattern:

-Length

-Data[Length]

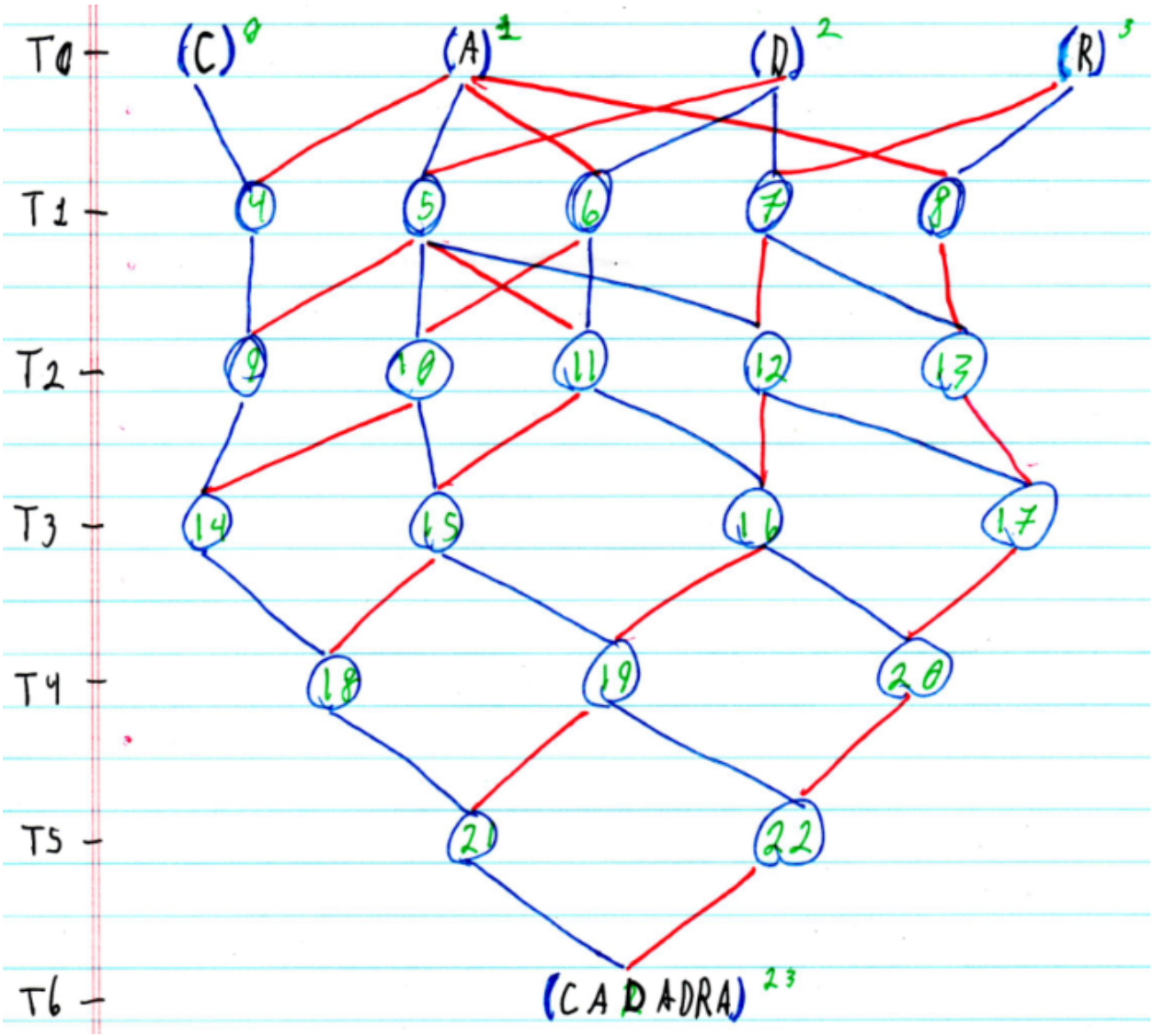
Although the visual display of the networks looks good it is cumbersome to work with in a practical sense. What is an easy way to work with data? Why tables of course, everyone loves tables. We can represent each node as a row in a table, each cell holds a bit of the nodes info, and arrange these rows into tables representing tiers. These tables are then put together in a book that forms the bulk of a network. There is a table for the network as well, which contains information such as the number of tiers in the network and such.

Here is the format for a state table followed by the format for a normal tier.



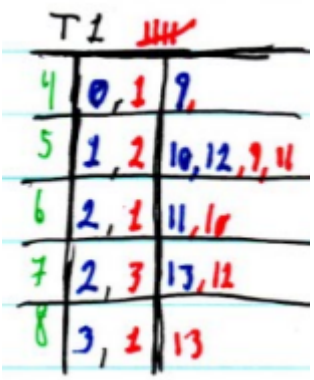
Input is the tier 0 state node table, and the other tier tables are the normal nodes. For the state table the NID is the node ID, STATE is the unit of data contained in the node, and Ul and UR are the upper left and upper right connections respectively. In the normal node tiers the TIER is the tier that the table represents, NID is the node ID, Ll is the lower left legs, Lr is the lower right legs. Ul is the upper left connections; Ur is the upper right connections.

For the examples following we will be using this network;

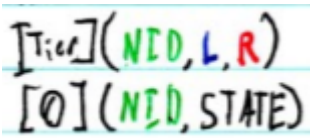


In this example tier [1] would look like this;

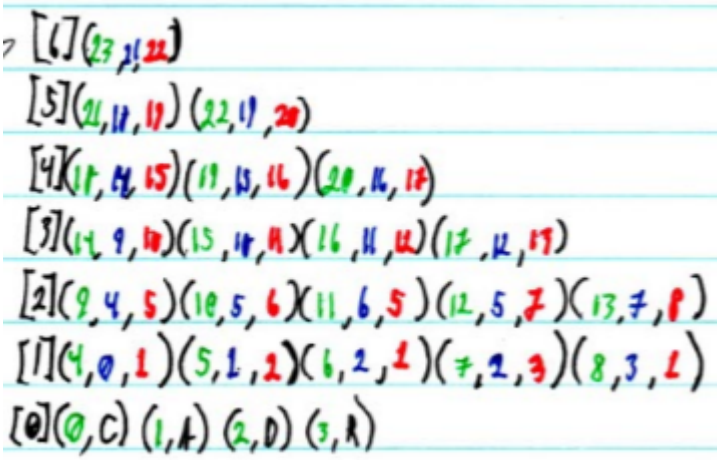
Tier:1 Node_Count:6
NID:4 Ll:0 Lr:1 Ul:9 Ur:
NID:5 Ll:1 Lr:2 Ul:10, 12 Ur:9, 11
NID:6 Ll:2 Lr:1 Ul:11 Ur:10
NID:7 Ll:2 Lr:3 Ul:13 Ur:12
NID:8 Ll:3 Lr:1 Ul: Ur:13



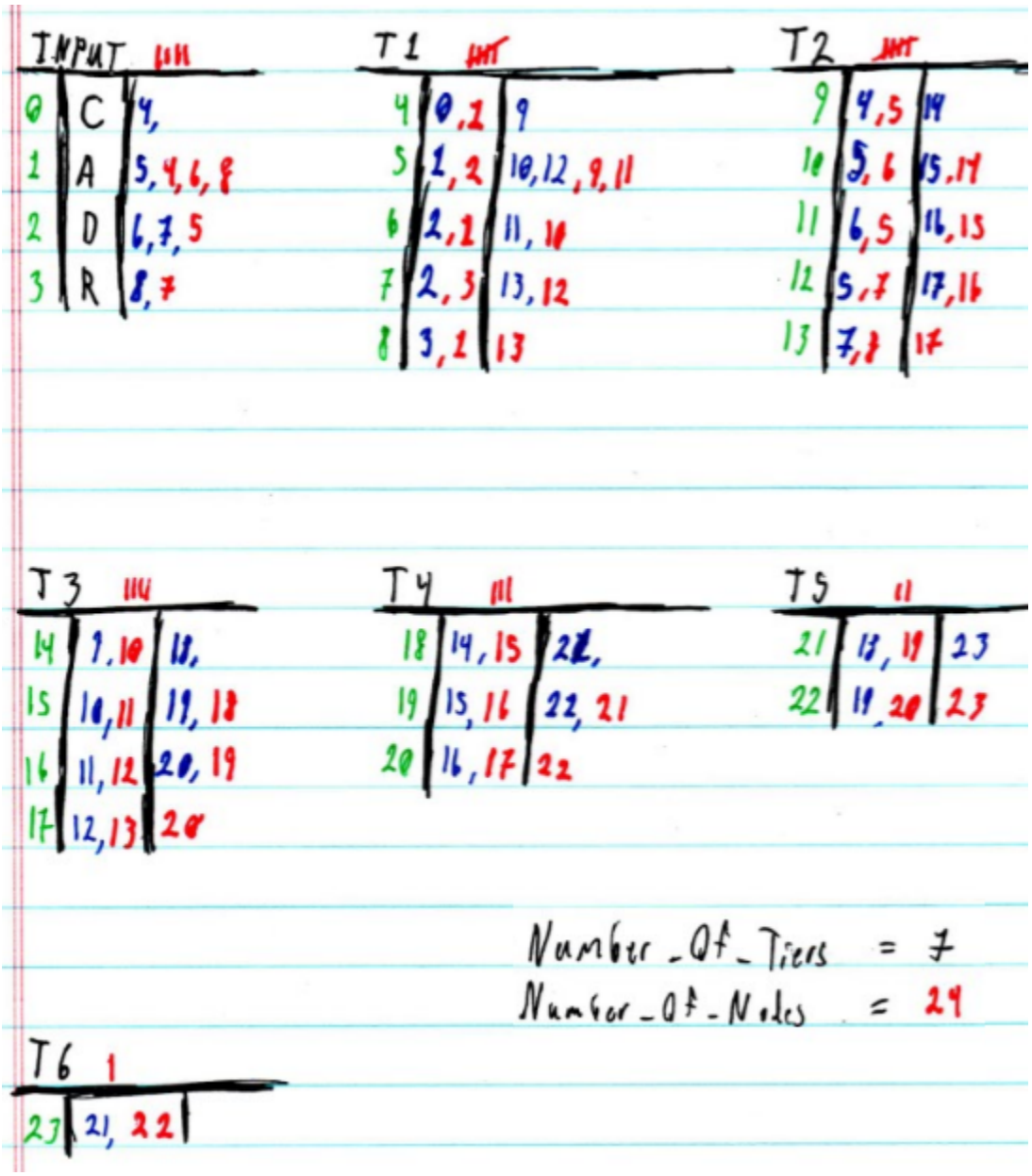
A more compact format would be showing the nodes but without the upper connections. This method is good for compact output of networks but terrible for tracing connections because you can only work top down without searching the entire upper tier. For demonstration purposes this works fine though as we can see the two dimensional representation, and assuming a relatively small network it is quite easy to figure out what is what.



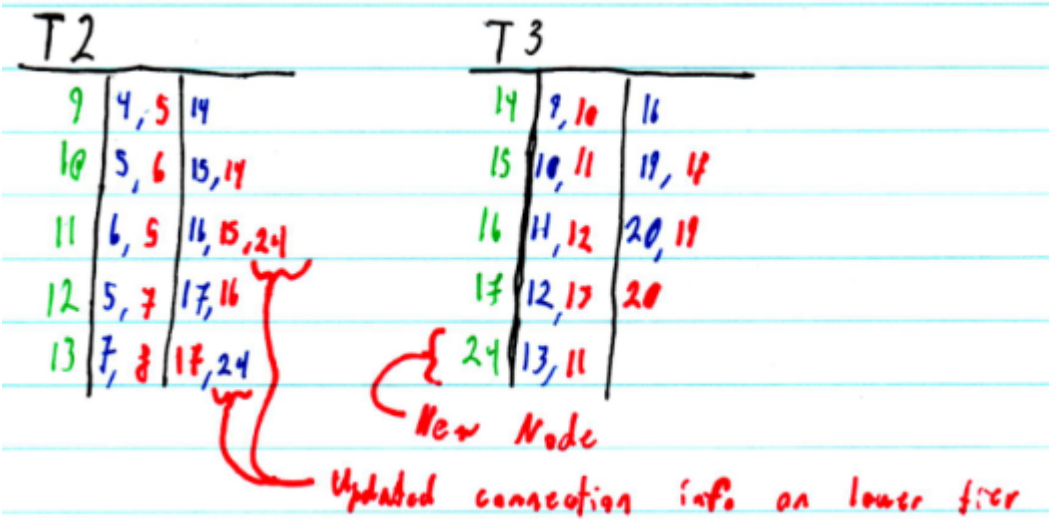
Now using this more compact method the entire network looks like this.



Now that we have explored a slightly more compact method of outputting nodes let us ignore it while resuming the slightly more verbose output. The entire network output using the tables appears as such;



Now that we have the network all laid out before us we shall attempt to add a node to this network. This is as simple as figuring out the new nodes information such as connections and such then updating the tables to reflect this new node and its connections and or state. For this example we will add the node (NID:24, LL:13, LR:11) to tier [3].



Adding a single node is relatively simple, but what if we want to add more than just a single node? Perhaps a whole new input set? Well let us attempt to work in the new input pattern “CAM” into the example network.

Now this will require a bit more work than before, we will have to take several steps to achieve this goal of adding a new input set to the network. So here are the steps we will be required to take on our journey to an integrated pattern;

0: Break the input pattern “CAM” into units of data or states.
1: Starting at the first index work through the input matching each state to a node.
2: When the entire pattern is read into the state nodes start at the first node in the state tier.
3: From the first node we check each set consecutively for an upper tier node.
4: If an upper tier node is found record it and move on, otherwise create it and move on.
5: Move up a tier and repeat the process of finding upper tier nodes.
6: Repeat steps 4 and 5 until you reach a tier with only one node.

Well now this seems all good and well, however, there is one major thing to note that we are skipping over here. When building these networks we need to keep track of the nodes as they are built, which we can do by building a temporary table to hold these nodes as we find and create them. We will call this table the current active node table (CAN).

When creating this temporary table it is easy to calculate the size needed because a networks size is predictable. When building an input the network that represents that particular input from tier 0 to the top tier is called a construct. These constructs have a number of tiers that is equal to the length of the input due to each tier having one less node than the tier before it.

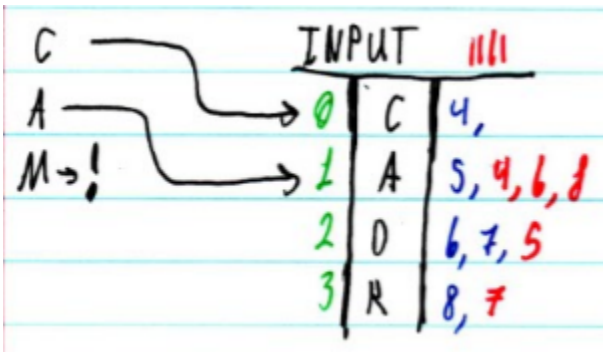
So we will write the CAN table using the following format:

[TIER]Index(NID)Index(NID)…

Using this format let's build a table for the example input "CAM". We will use the series of steps we detailed above to do so. So step one is to break this input into its component states. "CAM" = "C" + "A" + "M". Now we can see the input states, and we see the input is 3 units of data deep. So we can build the CAN to be filled in. For demonstration we will include the state in this example.

[0] 0("") 1("") 2 ("") //"" represents the state when found.
[1] 0( ) 1( )
[2] 0( )

Now we reference the raw tier table from our example network to see if the states in our example input have corresponding nodes.

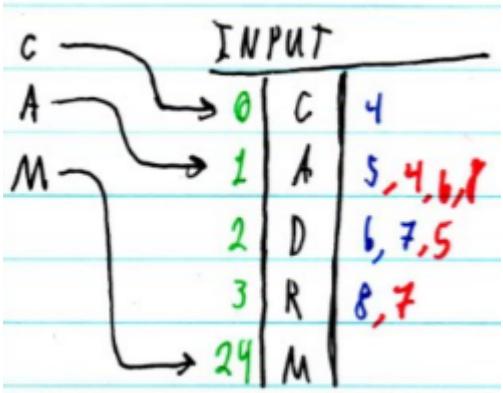


We start at the 0 index of the input "CAM" which is "C". Looking at the table we find that there is a node containing the state "C" with an ID of 0. So we can record this in our CAN changing the raw tier to [0] 0(0,"C") 1("") 2(""). Next we move down the input by one index finding "A" to correspond to the node with an ID of 1. Now the raw tier in the CAN looks like so; [0] 0(0,"C") 1(1,"A") 2("").

Moving on to the next input we find ourselves with "M". Looking to the network with the question does state node exist with input "M" we find that the answer is no. This is no problem; we have already covered how to add nodes to the network so we begin our work.

The current_Number_Of_Nodes in the network is equal to 24 at this point, now this count is kind of tricky for those not used to counting from 0 and referencing indexes. When the node count is at 24 this means that the current highest NID is 23. I will not explain why, if you need this answer look to the internet.

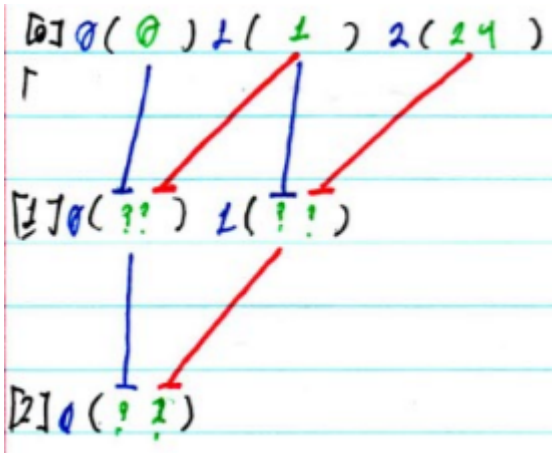
Now that we have the current number of nodes we create a node with the ID of one greater than the current greatest of 23. So our new node has a NID of 24. Then we increment the current_Number_Of_Nodes seeing as we just added a new one. After adding the node we set the state to "M" and get the new raw tier looking like so.



With this node we can now fill out the raw tier so the resultant CAN looks like this;

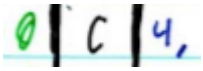
[0] 0(0, “C”) 1(1, “A”) 2 (24, “M”)
[1] 0( ) 1( )
[2] 0( )

Now that we have this tier we can move onto the next tier, tier [1]. We know that the nodes on the next tier will have a left and right leg linking the nodes on this tier together.



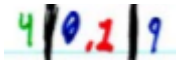
(0, 1)

Now we can see that we need to find a node with the lower connections for the first node in [1] because of the nodes in the [0] and [1] positions on the [0] tier. Referencing the raw tier table we can see the following for the node 0.

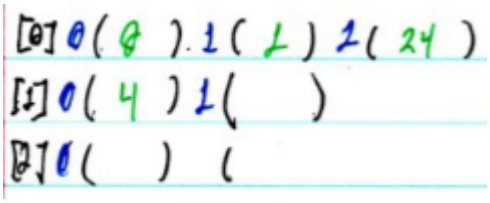


We can see from this that the node has an upper tier connection on the left leg, which for this couplet is what we are looking for because the “C” node is, the left portion of the “CA” pattern. So we

follow the connection up to see if node 4 has a right leg matching the one we are looking for which is node 1.



Lo and behold it does! The right leg matches the ID of the node we are looking for. So now we can fill in the CAN node on the next tier.

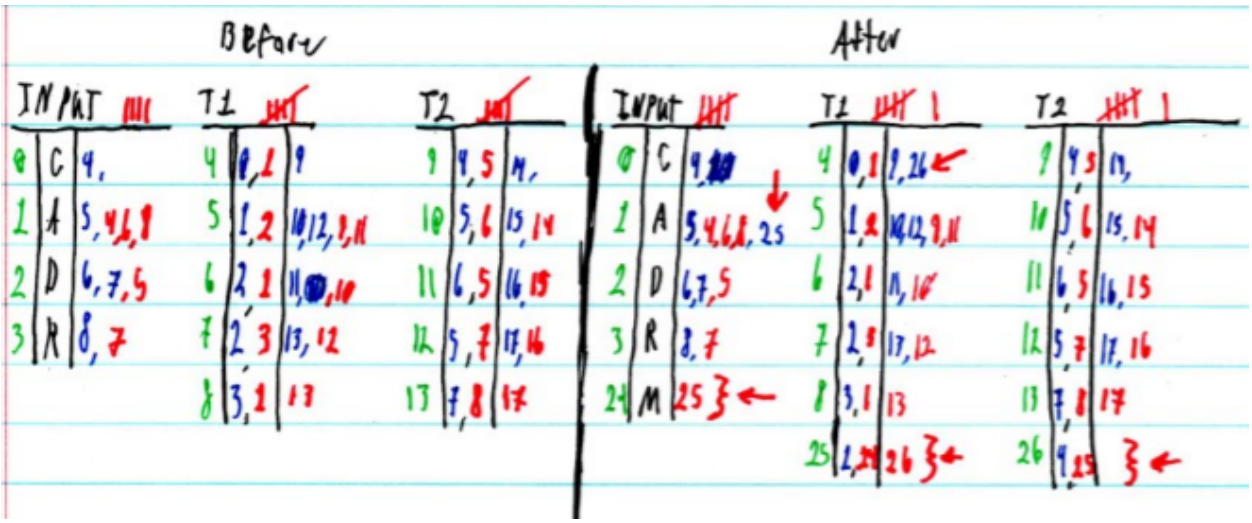


If we track the current node we are on with the variable Current_Node, which right now is 0 because we are on the first index of the input tier, then the node to fill in is on the tier (Current_Tier + 1) at index Current_Node. The nodes that we are currently checking for are in the indexes Current_Node and (Current_Node + 1). So when we move to the next input our Current_Node will be 1 and the nodes to check for a connection between are in indexes (1) and (1 + 1).

So now we move on to the next input and find that we are looking for nodes NID:1 and NID:24. Knowing that NID:24 was just created we are sure that no connections yet exist, so we can just create one. We do the same procedure as before with incrementing Number_Of_Nodes and such, but instead of setting a state in the node we instead set the lower connections.

Now we have completed tier 0 and move up to the next tier. On tier 1 we are checking NID:4 and NID:26, which was just created, and we create the node to link them being that there is currently no linking upper tier node. This completes tier 1.

Each tier is reduced by one node meaning the top tier will only have one node, which means there is no reason to evaluate that tier. With only one node on that tier there will be no upper connections. So the number of tiers we have to walk through is actually equal to (Input_Length – 1). The following image demonstrates the before and after of the building of the “CAM” construct on the network tables.



Before continuing we can delve a little deeper into the process of checking for upper tier nodes. The process is illustrated as follows;

get_Upper_Tier_Connection(p_Left, p_Right)
{
-Gather the two nodes that are to be linked, these will be called p_Left and p_Right.
-Does p_Left have and upper left connections?
~Yes
-Loop through these connections
-Does the connected upper node have a lower right matching p_Right?
~Yes
-Node is found, done.
~No
-Move on to the next connection
~No
-Create the new upper tier node with lower legs p_Left and p_Right
-This is the node needed, done.
-If the node was not found in the upper connections and one was not created create it now.
-Done, this new node is the one to use.
}

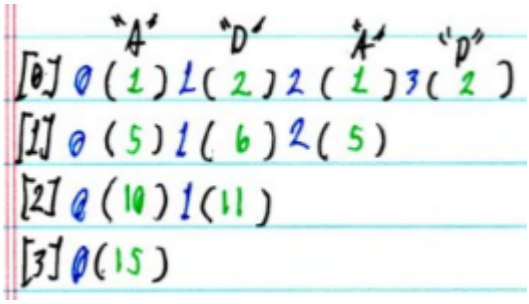
In the future we can refer to this little chunk of login with the function handle get_Upper_Tier_Connection(p_Left, p_Right). This function is to return the node when found, or create it if need be and return the newly created node. Either way we get a node with the appropriate lower connections.

We can lay out the process of gathering a state node in much the same way as above, let us use the function definition get_State_Node(p_State), with p_State being the unit of data to look for.

get_State_Node(p_State)
{
-loop through the state nodes
	-Does the current nodes state match p_State?
		~Yes
			-Found the node, done.
		~No
			-Continue on
-if no nodes has been found then create a new one now
-set the state of the node to p_State
-return the newly created node, we are done here
}

Now that we can build patterns we may go about collecting some, but there is an issue if you encounter the same pattern twice. If you build a pattern twice it does not create new connections or modify the network in any way. Which if you want to keep track of the commonality of the patterns, for example how many time a word appears in a book, this method is currently insufficient.

The answer to this problem is quite simple though. We simply attach a counter to each node. Every time this node is "built" the counter is iterated tracking how many times it has been accessed. Then the question is how to pick the nodes to iterate when building. This is done simply by utilizing the CAN we constructed when building the construct. Here is an example CAN.



Simple go through the CAN and tell each node to iterate its counter. There is a simple logic error here if you start your nodes with a reinforcement counter (RC) value of 1. When you go through the network it will increment the newly created nodes claiming its RC is 2. This is avoided easily by starting each node at RC 0, or by checking whether or not the node is new before iterating. Here is an example network with the effects of the reinforcement shown.

T	I	NID	RC
0	0	1	0+1=1
0	1	2	0+1=1
0	2	1	1+1=2
0	3	2	1+1=2

T	I	NID	RC
1	0	5	0+1=1
1	1	6	0+1=1
1	2	5	1+1=2

T	I	NID	RC
2	0	10	0+1=1
2	1	11	0+1=1

T	I	NID	RC
5	0	15	0+1=1