Excursions in Computing Science: Week 10a. Self

T. H. Merrett*
McGill University, Montreal, Canada

March 27, 2026

I. Prefatory Notes

1. Integration, This is the only set of Notes in the follow-up to this course CS199 in which I have not attempted to write up other people's well-established work, but am conducting an original investigation, however unsatisfactory. It is an attempt to characterize, not "consciousness", nor even "sentience", terms which I do not know how to define, but "self", which I consider to be more fundemental.

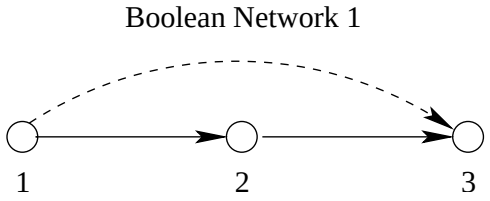
An entity has a *self* in the circumstance that the part will act on behalf of the whole, or the whole will act on behalf of the part. For example, an amoeba contracts to avoid acid, or fly kicks legs to turn over.

This characterization implies *integration* in the sense that wholes and parts communicate, and even that "whole" is a meaningful term.

It might even be possible to say that every living thing has a self. Integration can then be, for instance, via neurons for animals, or via the immune system for all life. Or it could be via pheromones for a hive, suggesting that there can be different levels of self, at different levels of integration.

So we need a notion of integration which abstracts from all these possible mechanisms. I am going to propose *transitive closure*.

2. Transitive closure. A *graph* is a mathematical object consisting of *nodes* which are connected by *links* or *edges*. Here is an example, which could abstract from three neurons connected together.



Neuron 1 connects to neuron 2 which in turn connects to neuron 3. Thus, indirectly, neuron 1 connects to neuron 3, as shown by the dashed line. (This dashed line is not a link in the graph,

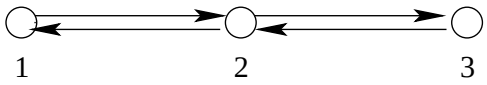
*Copyleft ©T. H. Merrett, 2026. Permission to make digital or hard copies of part or all of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage and that copies bear this notice and full citation in a prominent place. Copyright for components of this work owned by others than T. H. Merrett must be honoured. Abstracting with credit is permitted. To copy otherwise, to republish, to post on servers, or to redistribute to lists, requires prior specific permission and/or fee. Request permission to republish from: T. H. Merrett, School of Computer Science, McGill University, fax 514 398 3883. The author gratefully acknowledges support from the taxpayers of Québec and of Canada who have paid his salary and research grants while this work was developed at McGill University, and from his students and their funding agencies.

but a *path* which can be followed through the graph.)

But the connectivity is not complete. Neuron 3 does not connect back to neuron 2, and neither 2 nor 3 connect to 1. The graph is not fully integrated.

One way towards complete integration is by two-way ("bidirectional") links.

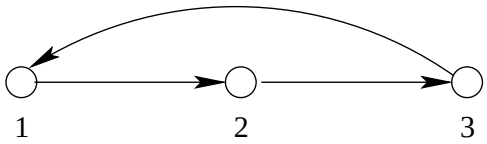
Boolean Network 2



If we were to show the paths in this graph, everything is connected to everything else.

Another way is to complete the cycle.

Boolean Network 3



3. Programming transitive closure. The *transitive closure* of a graph is a graph on the same nodes but with a link for every path in the original graph. In particular, it contains the original graph.

Since we must deal with hundreds—and in many cases billions—of neurons, we need a more powerful approach than just drawing pictures. Python is not the best programming language for this but it is readily available and my examples will be small. (For a Python tutorial, see my *Computing for teenagers*.)

We will represent a graph as a matrix, and in this case a Boolean matrix, having elements 0 and 1 (false and true). The three graphs above are respectively represented:

$$\begin{pmatrix} 0 & 1 & 0 \\ 0 & 0 & 1 \\ 0 & 0 & 0 \end{pmatrix}$$

In `Python`

```
TCB1 = [[0,1,0],[0,0,1],[0,0,0]]
```

$$\begin{pmatrix} 0 & 1 & 0 \\ 1 & 0 & 1 \\ 0 & 1 & 0 \end{pmatrix}$$

In `Python`

```
TCB2 = [[0,1,0],[1,0,1],[0,1,0]]
```

$$\begin{pmatrix} 0 & 1 & 0 \\ 0 & 0 & 1 \\ 1 & 0 & 0 \end{pmatrix}$$

In `Python`

```
TCB3 = [[0,1,0],[0,0,1],[1,0,0]]
```

You can check out how a 1 in row j, column k represents a link from neuron j to neuron k.

The links in the graph can be considered to be the paths of length 1. To find the paths of length 2, the matrix representation is handy: just multiply the matrix by itself. (For a tutorial on matrix

multiplication, see the beginning of my *Relativistic Quanntum Physics*.)

$$\begin{pmatrix} 0 & 0 & 1 \\ 0 & 0 & 0 \\ 0 & 0 & 0 \end{pmatrix}$$

In `Python`

```
prodBool(TCB1,TCB1)
[[0,0,1],[0,0,0],[0,0,0]]
```

Except that the elements are Boolean. If we treat them as numbers then the second, bidirectional, example yields (TCB2×TCB2)

$$\begin{pmatrix} 1 & 0 & 0 \\ 0 & 2 & 0 \\ 0 & 0 & 1 \end{pmatrix}$$

telling us that there are two ways of reaching the middle neuron by paths of length 2.

So our routine `prodBool` uses Boolean **and** instead of multiplying, and Boolean **or** instead of adding:

```
def prodBool(a,b):
 """like np.dot() but  and  for  * ,  or  for  +                THM 260104 """
 c=[]
 m=len(a)
 n=len(a[0])            # same as len(b)
 p=len(b[0])
 for j in range(m):
  c.append(range(p))
  for l in range(p):
   ab=0
   for k in range(n):
    ab=ab or (a[j][k] and b[k][l])
   c[j][l]=ab
 return c
```

But we want *all* paths up to the longest possible length, which may equal the total number of nodes in the graph (well, 1 less).

To find paths of length 3, we must combine length-1 with length-2. The straightforward way of doing this is to take the union of the two sets of paths, which we can do with a Boolean **or** of the two matrices:

```
def mator(a,b):
 """Boolean  or  of two same-size matrices                      THM 260104"""
 c=[]
 m=len(a)               # assume same len(b)
 n=len(a[0])            # assume same len(b[0])
 for j in range(m):
  c.append(range(n))
  for k in range(n):
   c[j][k] = a[j][k] or b[j][k]
 return c
```

These combine to give a single step of the transitive-closure algorithm:

```
def tcBstep(arrB,prevTC):
 """Accumulate next power of  arrB: start tcStep(arrB,arrB)       THM 260104
    uses prodBool(arrB,tc), mator(arrB,prod)"""
```

```
 prod=arrB
 tc=prevTC
 prod=prodBool(arrB,tc)
 tc=mator(arrB,prod)
 return tc
```

Let's see how this produces the transitive closure in each case.

Boolean Network 1

```
>>> TCB1 = [[0,1,0],[0,0,1],[0,0,0]]
>>> prevTC = TCB1
>>> prevTC=tcBstep(TCB1,prevTC)
>>> prevTC
[[0, 1, 1], [0, 0, 1], [0, 0, 0]]
>>> prevTC=tcBstep(TCB1,prevTC)
>>> prevTC
[[0, 1, 1], [0, 0, 1], [0, 0, 0]]
```

Boolean Network 2

```
>>> TCB2 = [[0,1,0],[1,0,1],[0,1,0]]
>>> prevTC = TCB2
>>> prevTC=tcBstep(TCB2,prevTC)
>>> prevTC
[[1, 1, 1], [1, 1, 1], [1, 1, 1]]
>>> prevTC=tcBstep(TCB2,prevTC)
>>> prevTC
[[1, 1, 1], [1, 1, 1], [1, 1, 1]]
```

Boolean Network 3

```
>>> TCB3 = [[0,1,0],[0,0,1],[1,0,0]]
>>> prevTC = TCB3
>>> prevTC=tcBstep(TCB3,prevTC)
>>> prevTC
[[0, 1, 1], [1, 0, 1], [1, 1, 0]]
>>> prevTC=tcBstep(TCB3,prevTC)
>>> prevTC
[[1, 1, 1], [1, 1, 1], [1, 1, 1]]
>>> prevTC=tcBstep(TCB3,prevTC)
>>> prevTC
[[1, 1, 1], [1, 1, 1], [1, 1, 1]]
```

In each case, the final iteration repeats the one before: all the paths have been found.

But in the last two case, the transitive closure is a *complete* graph: every node connects directly to every other. These are the cases which we can consider to be fully integrated, and what we might look for in a biological "connectome".

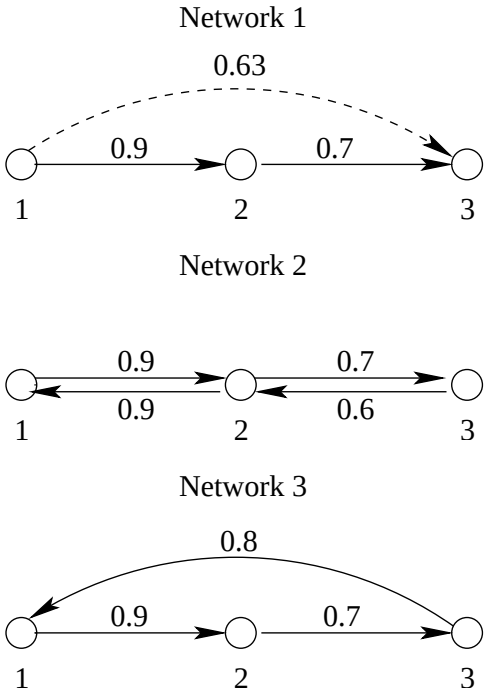
So we'll need a routine to find the smallest element in the transitive closure and check if it is 1.

```
def matminabs(mat):
 """Find smallest absolute value in matrix                    THM 251208
    Return first found and its location"""
 m=len(mat)
 n=len(mat[0])
 absmin=float('inf')            # anything is smaller than infinity
 for j in range(m):
  for k in range(n):
   amatjk=abs(mat[j][k])
   if amatjk<absmin:
    absmin=amatjk
    jmin=j
    kmin=k
```

```
 return absmin,jmin,kmin
if matminabs(...)[0]==1 ..
```

4. Networks. Biological connectomes are unfortunately not just graphs. The connections have *weights*, which we can take to be numbers, associated with each edge. Such constructions are called *networks*.

Here are three examples, variants on the Boolean graphs above. The first shows the length-2 path (dashed line) with a weight computed as the product of the two edge-weights: we will combine in this way, but alternatives are possible[1]



For numerical networks we replace the Boolean matrix multiplication with ordinary matrix multiplication (implemented in Python with `dot()` from the `numpy` library) and the Boolean matrix **or** with ordinary matrix sum.

```
def tcStep(arr,prevTC):
 """Accumulate next power of  arr: start tcStep(arr,arr)          THM 251231"""
 import numpy as np     # numerical python library
 mat=np.array(arr)
 tc=np.array(prevTC)
 prod=np.dot(mat,tc)    # matrix "dot product"
 tc=mat+prod
 return tc
```

[1] An elegant approach, which short-circuits most of this section, is to generalize the Boolean, replacing **and** with `min()` and **or** with `max()`. It may also be useful.
A *flow network* would require `min` and +, respectively for flows in series and in parallel.
Or $r_1 + r_2$ and $1/(1/r_1 + 1/r2)$, respectively, for electrical resistors in series and parallel.
Or $p_1 \times p_2$ and $1 - (1 - p_1) \times (1 - p_2)$, respectively, for (independent) probabilities,
None of these is good for links with negative weights, as might arise from inhibitory neurons.

Here is what results for the three examples.

Network 1

```
>>> TCN1 = [[0,0.9,0],[0,0,0.7],[0,0,0]]
>>> prevTC = TCN1
>>> prevTC=tcStep(TCN1,prevTC)
>>> prevTC
array([[ 0.  ,  0.9 ,  0.63],
       [ 0.  ,  0.  ,  0.7 ],
       [ 0.  ,  0.  ,  0.  ]])
>>> prevTC=tcStep(TCN1,prevTC)
>>> prevTC
array([[ 0.  ,  0.9 ,  0.63],
       [ 0.  ,  0.  ,  0.7 ],
       [ 0.  ,  0.  ,  0.  ]])
```

Network 2

```
>>> TCN2 = [[0,0.9,0],[0.9,0,0.7],[0,0.6,0]]
>>> prevTC = TCN2
>>> prevTC=tcStep(TCN2,prevTC)
 prevTC
array([[ 0.81,  0.9 ,  0.63],
       [ 0.9 ,  1.23,  0.7 ],
       [ 0.54,  0.6 ,  0.42]])
>>> prevTC=tcStep(TCN2,prevTC)
>>> prevTC
array([[ 0.81 ,  2.007,  0.63 ],
       [ 2.007,  1.23 ,  1.561],
       [ 0.54 ,  1.338,  0.42 ]])
```

Network 3

```
>>> TCN3 = [[0,0.9,0],[0,0,0.7],[0.8,0,0]]
>>> prevTC = TCN3
>>> prevTC=tcStep(TCN3,prevTC)
>>> prevTC
array([[ 0.  ,  0.9 ,  0.63],
       [ 0.56,  0.  ,  0.7 ],
       [ 0.8 ,  0.72,  0.  ]])
>>> prevTC=tcStep(TCN3,prevTC)
>>> prevTC
array([[ 0.504,  0.9  ,  0.63 ],
       [ 0.56 ,  0.504,  0.7  ],
       [ 0.8  ,  0.72 ,  0.504]])
>>> prevTC=tcStep(TCN3,prevTC)
>>> prevTC
array([[ 0.504 ,  1.3536,  0.63  ],
       [ 0.56  ,  0.504 ,  1.0528],
       [ 1.2032,  0.72  ,  0.504 ]])
```

For the first example we get the expected result, and it repeats at the next iteration.

The second example shows that there is no convergence: this is not really a transitive closure. The first result is expected: the paths from 1 to 3 and back give, respectively, $0.63 = 0.9 \times 0.7$ and $0.54 = 0.6 \times 0.9$; the pairs of paths that give the self-loops for each node, on the diagonal, give $0.81 = 0.9 \times 0.9$, $0.42 = 0.6 \times 0.7$, and, in the middle, the sum of these $1.23 = 0,81 + 0.42$.

But we must think about the result of the next iteration to find that it is describing paths of length 3 together with the shorter paths. For instance, $2.007 = 0.9 \times (0.81 + 0.42 + 0.9 = 0.9 \times 1.23 + 0.9$.

(Note that, since our initial weights—on the edges of paths of length 1—all have 1 decimal place, the number of decimal places in each result is the maximum path length contributing to it.)

What is happening is that the self-loops make the maximum path length grow without limit as we make further iterations.

To stop this unlimited growth we must stop the iterations once we have reached the transitive closure. To do that requires cooperating with the (Boolean) transitive closure routine we started with.

We will use the network matrix but derive a Boolean matrix from it:

```
idef toBool(a):
 """Convert matrix to Boolean                                    THM 260104"""
 b=[]
 m=len(a)
 n=len(a[0])                    # assume rectangular: same for all rows
 for j in range(m):
  b.append(range(n))
  for k in range(n):
   b[j][k] = a[j][k] and 1 # pythoon treats a[j][k] nonzero as Boolean 1
 return b
```

Here is the code, combined with our earler transitive closure.

```
def integrated(mat):
 """Report how integrated is the network represented by matrix  mat  THM 260106
    Technique: parallel boolean, real transitive closures until boolean says
    fully closed, then report smallest value in real t.c.
    uses toBool(),matminabs(mat)[0],tcStep(mat,prevTC),tcBstep(matB,prevTCB) """
 matB=toBool(mat)
 n=1
 nmax=len(mat)          # assume square matrix; max pathlength is nmax
 leastB=matminabs(matB)[0]
 least=matminabs(mat)[0]
 if leastB==1:
  return n,least
 prevTCB=matB
 prevTC=mat
 while leastB<1:        # it's not a full transitive closure
  n=n+1
  if n>nmax:            # not a complete graph
   return n,0
  prevTCB=tcBstep(matB,prevTCB)
  prevTC=tcStep(mat,prevTC)
  leastB=matminabs(prevTCB)[0]
  least=matminabs(prevTC)[0]
  return n,least,momentsAbs(prevTC)
```

And here are the results for the three examples.

```
>>> integrated(TCN1)
(4, 0)
>>> integrated(TCN2)
(2, 0.41999999999999998, (9.0, 0.74777777777777776, 0.24273327840336323, 1.23))
>>> integrated(TCN3)
(3, 0.504, (9.0, 0.64688888888888885, 0.1435099686820093, 0.90000000000000002))
```

The first two numbers give a measure of integration. The first is the number of iterations to reach transitive closure: if it exceeds the number of nodes, the iterations were stopped without reaching closure. The second ia the smallest entry in the numerical matrix corresponding to the transitive closure: the larger it is relative to the initial entries, the more integrated the network seems to be.

The remaining four numbers are supplementary statistics on the matrix elements: count, average, standard deviation and maximum value. The code is

```
def momentsAbs(mat):
 """cslculate moments 0,1,2 in rectangular numeric array and from that one pass
   return count, average, standard deviation, maximum           THM 260122"""
 m=len(mat)
 n=len(mat[0])          #assume rectangular so this is same for all sublists
 zeroth=0
 first=0
 second=0
 most=0
 for j in range(m):
  matj=mat[j]
  for k in range(n):
   matjk=abs(matj[k])
   zeroth=zeroth+1      #should sum to m*n
   first=first+matjk
   second=second+matjk**2
   if matjk>most:
    most=matjk
 count=zeroth+0.0       #make real
 avg=first/count
 stdev=np.sqrt((second-2*first*avg+count*avg**2)/(count-1))
 return count,avg,stdev,most
```

5. The Neural Self. Fortunately there has been recent work on biological "connectomes", whose integration we can explore.

Randi et al. [RSDL23] measured signal propagation in the planaria flatworm C. elegans and published a 168-by-168 matrix of connections.

Running `integrated` on a copy of this data formatted for Python gave closure in 2 iterations. It seems the flatworm has a self:

```
integrated(Celegans)
(2, 1.7824861769699818e-05,
 (28224.0, 0.21485800094182647, 0.23004576221224274, 6.2006432041071653))
```

6. The Immunological Self. Things did not go so smoothly for the human immune system, as published by Shilts et al. [SSea22]. Their Fig.1e shows a network which is clearly disconnected into many components and so cannot produce a complete transitive closure. Manually extracting the data for the one major connected component, and formatting it for Python, produced a 124-by-124 double matrix: each link had two weights, usually but not always complementary (sometimes both were nonzero), the second weight being either 0 or 1. I summed these to get a single weight for each link.

Running `integrate` on the result did not produce a complete closure.

Instead of being scientific and abandoning the hypothesis that selfhood entails a complete transitive closure, I consider that my selection of protein linkages in the immune system is not relevant.

The protein binding interactions investigated could be entirely local and not reflect communication among remote parts of the body, in which case we need some other aspect of the immune system.

7. The Pheromonal Self. On "pheromone connectome", Google's AI says "Chemosensory connectomics is still in its early stages but holds promise for understanding the fundamental principles of social signaling, sexual behavior, and how environmental chemical cues impact development."

Adding the specification "bees" gets a list of 5 of over 50 known pheromones and their effect on

the hive.

Maybe a full-hive connectome will emerge soon.

References

[RSDL23] Francesco Randi, Anuj K Sharma, Sophie Dvali, and Andrew M Leifer. Neural signal propagation atlas of caenorhabditis elegans. *Nature*, 623(7956):406–14, Nov. 2023. PMCID: PMC10632145 PMID: 37914938; source data for Fig.2 at https://www.nature.com/articles/s41586-023-06683-4#MOESM5 (accessed March 2026).

[SSea22] J. Shilts, Y. Severn, and F. Galaway et al. A physical wiring diagram for the human immune system. *Nature*, 608(7922):397–404, Aug 2022. Source data for Fig.1e at https://www.nature.com/articles/s41586-022-05028-x#MOESM6 (accessed March 2026).