
Haerem, Pentland and Miller (2015) propose an extended construct of task complexity that incorporates the social and material context of a task by (a) representing the task as a network of events and (b) counting paths within the network. The number of paths (or ties within the paths) provides an index of task complexity. This document provides some methodological guidance on how to compute this measure of task complexity.

In theory, task complexity can be estimated by counting the number of paths in a network that represents a model of the task. However, counting paths in a network is known to be a “#P-complete” problem (Bax, 1994): the number of paths cannot be counted in polynomial time. As a practical matter, as the size of the network increases, no amount of computing resources can solve the problem. The number of paths can be enumerated directly for smaller networks (Rubin, 1978; Bax, 1994) and estimated for larger networks (Roberts & Kroese, 2007).

Software requirements
The code we provide here is compatible with MatLab R2014a. MatLab is a product of The MathWorks, Inc.

Computing the task complexity index
There are two major steps in computing the complexity of a task: (1) constructing the task network and (2) computing the index. We use the term “index” to reinforce the point that the resulting number is not an objective physical quantity, like mass or temperature. Rather, like the “consumer price index” or any other constructed measure, it provides an indicator of an underlying phenomenon that would otherwise be impossible to observe or quantify.

Step 1: Constructing the task network
Hærem et al (2015) describe the task network as follows:

... we model tasks as networks of events, where an event consists of an action performed by some actor at some moment in time. Events generate information cues that may (or may not) be processed by other actors. The actors may be people, machines, organizational sub-units (Latour, 2005). Pentland and Feldman (2007) proposed the use of this kind of network as a model for organizational routines and refer to the nodes as functional events.

Thus, the task network is a special kind of network where each node describes an action by a specific actor. As the number of actors involved in a task grows, so does the task network. Note that “actors” can include non-human actors, such as computer systems or physical objects that are part of the task context. This is how the concept of task complexity is
extended to incorporate the social and material context. When the task network is limited to an idealized set of actions by a single (implicit) actor, it corresponds to the approach described by Wood (1986).

The task network can be constructed from any source. One can start with idealized researcher insight, as in Wood (1986), or empirical observations. Either way, the creation of the network involves researcher judgment. Among other things, the researcher must decide:

- the boundaries of the task, including the actions, actors and artifacts that may be involved;
- the level of detail or granularity used to describe the task;
- the criteria for identifying ties between events;
- assumptions about source and terminal nodes (where the task begins and ends).

Each of these assumptions entails judgment on the part of the researcher, based on their research question and their understanding of the task domain. Each of these factors will have an impact on the size and structure of the task network and the resulting task complexity index.

**Step 2: Computing the complexity index**

Given a task network, one can count paths within the network, which provides an index of task complexity. For small task networks (with less than 10 nodes and 20 ties), it is possible to compute task complexity index by direct enumeration (e.g., “brute force”) using a typical microprocessor. For larger task networks, researchers with access to supercomputing facilities may have some success, but the computational time increases surprisingly fast. Above a certain size of network, the complexity index must be estimated.

**Counting paths and ties.** Hærem et al (2015) show that counting the ties along each path in the task network closely corresponds to the concept of “coordinative complexity” introduced by Wood (1986). Conceptually, ties between events indicate information cues or some other mechanism that connects one event to the next. Thus, the published definition is stated in terms of counting ties along each complete path. However, researchers need to be aware of some key issues in this seemingly simple computation:

1. **Identifying “source” and “terminal” nodes in the network.** In some task networks, there is a clearly defined starting point (“source”) and ending point (“terminal”). In other task networks, there may be multiple sources and terminals, and therefore many more paths. The code provided here requires a single specific source/sink; if there are multiple sources or sinks, the results can be added for each pair.

2. **Assumptions about cycles in the graph.** In particular, can steps be repeated or not? If not, then the graph is “acyclic” and counting paths is easier. If a task involves steps that can be repeated, then the task network has cycles. This vastly increases the number and length of pathways from source to terminal. In principle, including cycles allows an infinite path-length, and cycles have a multiplicative effect on the number of paths.
To understand how this works, it is useful to work through some simple examples. Harem et al (2015) provide examples that are based on idealized task descriptions, as shown here in Figure 1. The specific number of paths (and ties along each path) depends on how one defines the source nodes and terminal nodes for the task and whether one allows any repetition of the nodes.

Here, we show two different methods of counting. In the first method, we assume that each task has a single source and a single terminal node. In the second method, we assume that every node can be a source, so there are multiple source nodes and a single terminal node. Both of these methods allow some repetition of nodes, so technically these are not simple paths (where there is no repetition of nodes). Rather, they are “open walks,” from source to terminal, allowing a single repetition of each cycle. If we restricted ourselves to “simple paths”, without repetition, then the patrol boat and frigate would each have only 2 paths.

Notice that the total number of paths and ties changes depending on how the source and terminal nodes are defined, even though the task network is unchanged. However, the relative complexity of each role compared to the others does not change. With either method, “airplane” has lowest complexity index and “patrol boat” has the largest. There is no universal answer to this question; it will be different for different tasks. Researchers need to define sources, sinks, and assumptions about cycles according to their own particular research questions.

**Figure 1: Counting paths**

Assuming a single source node and a single sink node:

<table>
<thead>
<tr>
<th>AIRPLANE ONLY</th>
<th>Path</th>
<th>Ties</th>
</tr>
</thead>
<tbody>
<tr>
<td>a</td>
<td>3</td>
<td>1</td>
</tr>
<tr>
<td>Total</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>PATROL BOAT ONLY</th>
<th>Path</th>
<th>Ties</th>
</tr>
</thead>
<tbody>
<tr>
<td>a</td>
<td>4</td>
<td>1</td>
</tr>
<tr>
<td>b</td>
<td>6</td>
<td>1</td>
</tr>
<tr>
<td>c</td>
<td>3</td>
<td>1</td>
</tr>
<tr>
<td>d</td>
<td>5</td>
<td>1</td>
</tr>
<tr>
<td>Total</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>FRIGATE ONLY</th>
<th>Path</th>
<th>Ties</th>
</tr>
</thead>
<tbody>
<tr>
<td>a</td>
<td>3</td>
<td>1</td>
</tr>
<tr>
<td>b</td>
<td>5</td>
<td>1</td>
</tr>
<tr>
<td>c</td>
<td>2</td>
<td>1</td>
</tr>
<tr>
<td>d</td>
<td>4</td>
<td>1</td>
</tr>
<tr>
<td>Total</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

Assuming multiple source nodes and a single sink node:

<table>
<thead>
<tr>
<th>AIRPLANE ONLY</th>
<th>Path</th>
<th>Ties</th>
</tr>
</thead>
<tbody>
<tr>
<td>a</td>
<td>1</td>
<td>3</td>
</tr>
<tr>
<td>b</td>
<td>2</td>
<td>2</td>
</tr>
<tr>
<td>c</td>
<td>3</td>
<td>1</td>
</tr>
<tr>
<td>Total</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>PATROL BOAT ONLY</th>
<th>Path</th>
<th>Ties</th>
</tr>
</thead>
<tbody>
<tr>
<td>a</td>
<td>1</td>
<td>4</td>
</tr>
<tr>
<td>b</td>
<td>2</td>
<td>3</td>
</tr>
<tr>
<td>c</td>
<td>3</td>
<td>2</td>
</tr>
<tr>
<td>d</td>
<td>4</td>
<td>1</td>
</tr>
<tr>
<td>f</td>
<td>5</td>
<td>2</td>
</tr>
<tr>
<td>g</td>
<td>6</td>
<td>1</td>
</tr>
<tr>
<td>h</td>
<td>7</td>
<td>3</td>
</tr>
<tr>
<td>k</td>
<td>3</td>
<td>1</td>
</tr>
<tr>
<td>l</td>
<td>4</td>
<td>2</td>
</tr>
<tr>
<td>m</td>
<td>5</td>
<td>1</td>
</tr>
<tr>
<td>Total</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>FRIGATE ONLY</th>
<th>Path</th>
<th>Ties</th>
</tr>
</thead>
<tbody>
<tr>
<td>a</td>
<td>1</td>
<td>3</td>
</tr>
<tr>
<td>b</td>
<td>2</td>
<td>2</td>
</tr>
<tr>
<td>c</td>
<td>3</td>
<td>1</td>
</tr>
<tr>
<td>d</td>
<td>3</td>
<td>3</td>
</tr>
<tr>
<td>e</td>
<td>4</td>
<td>2</td>
</tr>
<tr>
<td>f</td>
<td>5</td>
<td>1</td>
</tr>
<tr>
<td>g</td>
<td>6</td>
<td>1</td>
</tr>
<tr>
<td>h</td>
<td>2</td>
<td>1</td>
</tr>
<tr>
<td>i</td>
<td>2</td>
<td>3</td>
</tr>
<tr>
<td>j</td>
<td>3</td>
<td>2</td>
</tr>
<tr>
<td>k</td>
<td>4</td>
<td>1</td>
</tr>
<tr>
<td>Total</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>
Enumerating paths in smaller networks. Even in a small network, there can be thousands of paths, and it is necessary to use a computer program to count them. However, since the underlying problem is \#P-complete, it is only feasible for smaller networks. We provide two examples of code for this computation here.

Appendix A includes code written to implement the algorithm described in Haerem et al (2015). This algorithm counts paths in two passes. First, it counts the paths created by branching. Then, it counts the paths created by loops. It counts the effect of each loop once.

Branches. In an adjacency matrix, edges in the upper right triangle (above the diagonal) represent branches. When there is more than one non-zero entry in the same row, it represent a branch. These paths are counted using a breadth first search. Partial paths are completed using the built-in function for finding the shortest path. This results in a list of “simple paths” that do not contain loops.

Loops. In an adjacency matrix, edges in the lower left triangle (below the diagonal) represent loops. When there is a non-zero entry in this section of the matrix, it means that the path can jump back to a prior position. The algorithm takes each loop and applies it to each of the simple paths. It counts every place the loop could be inserted; it counts the new path and adds the length of the new path to the total ties. The algorithm avoids running forever by counting each loops once. On a MacBook Pro, this algorithm can count the paths in a 15x15 adjacency matrix with over 100 edges in less than 60 seconds (over 50,000 paths, with over 7.95 million ties). Processing time is a strong function of the number of edges.

Appendix B includes code from Professor Or Zuk, Department of Statistics, Hebrew University of Jerusalem: https://github.com/orzuk/MatUtils/tree/master/math.2 We include it here for purposes of comparison. We have simplified the code for our specific application. The algorithm counts all of the paths less than or equal to a given length, from source to sink. While it does not count the number of ties, the results from this algorithm correlate well with our algorithm. It avoids running forever by limiting the length of the paths it counts. However, because it is counting all possible paths, including multiple iterations and combinations of every loop, it is limited to rather small networks (on the same MacBook Pro, a 10x10 adjacency matrix may not finish counting overnight).

Estimating paths in larger networks. As the task network grows, it is not possible to compute the number of paths in polynomial time, with or without a supercomputer. To address this problem, we offer two approaches. The first is an implementation of the algorithm provided by Roberts & Kroese (2007). This method works well for networks with 50 < n < 100 nodes and density \( d > 0.2 \). Such networks can have a large number of paths \( 10^{50} - 10^{150} \).

---

2 Professor Zuk’s web site states that: “Usage is free and at your own risk.”
http://pluto.huji.ac.il/~orzu/code.html. We have adapted it for our application here.
In working with some empirically derived task networks, we have found that some have lower density \((d < 0.1)\). For such networks, Roberts & Kroese’s (2007) sometimes yields implausible results (e.g., the number of estimated paths on the order of \(10^{13}\)).

To estimate the number of paths in lower density networks (which are probably typical of real task networks), we worked out a simple method that involves extrapolating from smaller networks to larger networks. The procedure involves three steps:

1. Compute the actual number of paths for the smaller networks in set of observed data using the code in Appendix A (or your own algorithm, if you prefer).
2. Fit an exponential function to the observed data, where the independent variables are the number of nodes, branches and loops:

\[
\ln(\text{paths}) = \beta_1 \text{(nodes)} + \beta_2 \text{(branches)} + \beta_3 \text{(loops)} + \beta_2 \text{(branches}^2) + \beta_3 \text{(loops}^2)
\]

3. Use this exponential function to extrapolate to networks where direct enumeration is not feasible.

**Summary.** The choices for computing the complexity index are summarized in Table 1. One of the important things to notice about Table 1 is that the task complexity index can vary by over 100 orders of magnitude, depending on the size and density of the task network. While there is clearly room for improvement in tools and strategies for this computation, one thing should be very clear: task complexity is not just “high” or “low”.

**Table 1: Alternative methods for computing the task complexity index**

<table>
<thead>
<tr>
<th>Method</th>
<th>Applicability</th>
<th>Task Complexity Index Range</th>
<th>See Appendix</th>
</tr>
</thead>
<tbody>
<tr>
<td>Direct enumeration</td>
<td>5 &lt; nodes &lt; 20 density &lt; 0.4</td>
<td>1 - 10^6</td>
<td>A</td>
</tr>
<tr>
<td>Extrapolation from small networks</td>
<td>5 &lt; nodes &lt; 50 density &lt; 0.2</td>
<td>10^6 - 10^{15}</td>
<td>C</td>
</tr>
<tr>
<td>Roberts and Krause (2007)</td>
<td>50 &lt; nodes &lt; 100 density &gt; 0.2</td>
<td>10^{20} - 10^{150}</td>
<td>D</td>
</tr>
</tbody>
</table>

**References**


Appendix A: Counting paths by direct enumeration (each loop once)

function [ total_ties, total_paths, simple_paths, loop_paths, nbranches, nloops ] = 
    task_complexity_index(adjMtx, source, sink)

%% count paths and ties by brute force
% BT Pentland June 2015
%
% This code implements the algorithm described in Haerem, Pentland and 
% Management Review.
%
% This code implements this algorithm by counting paths in two passes 
% 1) branches only (upper right) -> simple paths
%    This pass uses a breadth first search.
% 2) loops only (lower left, excluding diagonal) 
%    For each loop, Combine unique first/last parts of all simple paths 
%    This pass counts each loop once.
%
% INPUT:
% adjMtx: adjacency matrix that represent the task
% source: starting point for task
% sink: ending point for task
%
% OUTPUT:
% total_ties: total number of ties along the total_paths, as defined in AMR 
% total_paths: total number of paths from source to sink, counting each loop once.
% branch_paths: paths due to branching (from first pass)
% loop_paths: paths due to looping (from second pass)
% nbranches: number of edges in upper right (branches)
% nloops: number of edges in lower left (loops)

% NOTE:
% 1) This algorithm excludes the effects of self-connected nodes
% 2) For adjMtx larger than about 20x20, it may not be practical to use this 
% algorithm on a microprocessor
3) The thresholds on line 73 and lines 191 & 213 are intended to limit CPU time. They can be adjusted accordingly, or removed entirely.

% initialize these first...
   total_paths = 0;
   simple_paths = 0;
   loop_paths = 0;
   total_ties = 0;
   paths{1,1}=[];

% Use CAPITAL N for size of adjacency matrix
N = size(adjMtx,1);

% first make sure adjMtx is 0/1 only
adjMtx = (adjMtx >0);

% Count the branches and loops
branches = branches_only(adjMtx);        
loops = loops_only(adjMtx);

nbranches = numel(find(branches));
nloops = numel(find(loops));

disp(strcat('NBranches=',num2str(nbranches), ' NLoops=',num2str(nloops)));

% if the matrix is too big/dense, then estimate the number of total ties
% This threshold can be adjusted based on available computing capacity.
% 200 is a sensible limit for a microprocessor... but YMMV (your mileage may vary)
% The coefficients in this equation were estimated by fitting this
equation to simulated data, with 5 <= N nodes <= 20, and a probability
% of branches or loops ranging from 0 < p(edge) < 0.3. In the simulated
% data, branches and loops vary independently, and the total network density
% ranges between 0 <= density <=0.3. R-squared >=0.95 consistently on
% several trials.
if (nbranches + nloops) > 200

   total_ties = exp(2.5988 - 0.30909*N + 0.42036*nbranches + 0.23811*nloops -
                 0.0026285*nbranches^2 - 0.003174*nloops^2);

   disp('**TOO BIG -- ESTIMATING TIES**');

   return;
end
% First count all simple paths created by branching
% Use breadth first search, start with the source node

% Initialize the paths starting from the source
next_nodes = (branches(source,:) >0);  
next_nodes = find(next_nodes);  
for n_loop =1:numel(next_nodes)  
    paths{n_loop,1}= [source,next_nodes(n_loop)];  
end

% code below assumes that there are paths, so stop if there are not
if isempty(paths{1,1})
    disp('no simple paths - stopping')
    return;
end

% keep going...
for B=1:size(branches)
    for p = 1:size(paths,1)

        a = paths{p}(end): % last element of the path so far

        % get the list of out nodes for the current node a
        next_nodes = (branches(a,:) >0);
        % if there is only one, then just extend the current path
        n = find(next_nodes);
        if numel(n) == 1    
            paths{p} = [paths{p},n];
        else       % if there is more than one, then extend the current path
            % and add a new path for each
            if numel(n) >1
                path_so_far = paths{p};

                % extend current path using first branch
                paths{p} = [path_so_far,n(1)];

                % also add new paths for each additional branch
                % need to check if it is unique before storing it
                for j=2:numel(n)
                    if keep_path(paths, [path_so_far,n(j)])
                        paths=add_new_path(paths,path_so_far,n(j));
                    end
                end
            end
        end
    end
end
end

% finish any incomplete paths, which may come up in real data
g = sparse(adjMtx);
for p = 1:size(paths,1)
    if paths{p}(1,end) ~= sink
        [dist, shortestpath, pred] = graphshortestpath(g, paths{p}(1,end), sink);
        paths{p} = [paths{p}, shortestpath(2:end)];
    end
end

% if this method results in no paths from source to sink, the use
% the builtin MatLab function to find at least one path.
if isempty(paths)
    disp('no paths - adding shortest')
    [dist, shortestpath, pred] = graphshortestpath(g, source, sink);
    paths{1} = shortestpath;
    return;
end
if size(paths,1) == 0
    disp('no paths - stopping')
    return;
end

% count paths and ties so far
simple_paths = size(paths, 1);
for k=1:size(paths, 1)
    total_ties = total_ties + numel(paths{k});
end
% disp(strcat('done with branches. Simple paths = ',num2str(simple_paths)))

% now count paths created by loops
% For each loop, combine unique first/last parts of all simple paths
% Column index is the destination, row index is the start

% Apply each loop once to all the paths
% Just apply the loops to the initial set of paths
% This is conservative: loops are not combined.
for loop=find(loops)'

    % create cell arrays to store the intermediate paths
    root_paths=cell(1);
    branch_paths=cell(1);
    n_roots=0;
    n_branches=0;
%get the from/to for each loop
[loop_from, loop_to] = ind2sub([N,N],loop);

% Get the unique first parts = roots for new paths
for p=1: simple_paths % use size(paths,1) for combinatoric explosion

% just keep the first 1000 -- this is an arbitrary limit to prevent
% unlimited CPU usage
if n_roots >= 1000
    continue;
end

% find if there is a loop FROM this path
% store the starting point and the path so far
cut_idx = find(paths{p}==loop_from);
if cut_idx
    new_root_path = paths{p}(1:cut_idx);

    % need to check if it is unique before counting it.
    if keep_path(root_paths, new_root_path)
        root_paths = cat(1, root_paths, new_root_path);
        n_roots= n_roots+1;
    end
end
end

% %%%%%%%%%% Get the unique branch parts %%%%%%%%%%%%%%%%%%%%%%%%%%%%%
for p=1: simple_paths % use size(paths,1) for combinatoric explosion

% just keep the first 1000 -- arbitrary limit
if n_branches >= 1000
    continue;
end

% find if there is a loop INTO this path
% store from that point to the end
cut_idx = find(paths{p}==loop_to);
if cut_idx
    new_branch_path = paths{p}(cut_idx:end);

    % need to check if it is unique before storing it
    if keep_path(branch_paths, new_branch_path)
        branch_paths = cat(1, branch_paths, new_branch_path);
        n_branches= n_branches+1;
    end
end
end
% Combine unique root and branch paths and count the new new paths & ties
%     disp(strcat('New paths for loop=', num2str(loop_from),';-',num2str(loop_to), ', = ',
%     num2str(n_roots), 'x',num2str(n_branches)));

loop_paths = loop_paths + n_roots*n_branches;

for i=2:size(root_paths,1)
    for j=2:size(branch_paths,1)
        new_path = [root_paths{i}, branch_paths{j}];
        total_ties = total_ties+ numel(new_path);
    end
end
end

%% Now total it up = all done
total_paths = simple_paths + loop_paths;
% disp(strcat('done with loops. Loop paths = ',num2str(loop_paths)))
end

function paths = add_new_path(paths,so_far,rest)
    paths = cat(1,paths, [so_far,rest]);
end

function keep = keep_path(existing_fragments, new_fragment)
    keep=1;
    for k=1:size(existing_fragments,1)
        if isequal(existing_fragments{k}, new_fragment)
            keep=0;
            return;
        end
    end
end

function new = branches_only(a)
    n =length(a);
    new = zeros(n);
    for i=1:n
        for j=i+1:n
            new(i, j) = a(i, j);
        end
    end
end
function new = loops_only(a)

    n = length(a);
    new = zeros(n);
    for i = 1:n
        for j = 1:i-1
            new(i, j) = a(i, j);
        end
    end
end
Appendix B: Alternative method for counting paths

Key parts of this algorithm and code were downloaded from https://github.com/orzuk/MatUtils/tree/master/math.

% This function counts ALL of the paths (including multiple loops) between % the source and sink notes in a network. It finds sources & % sinks by examining the in-degree and out-degree of nodes in the network. % To apply different assumptions about sources, sinks, and cycles, the code % would need to be edited accordingly.

function c = count_graph_paths(adjMtx)
    sinks = get_graph_sinks(adjMtx);
    sources = get_graph_sources(adjMtx);
    [paths, num_paths_pairwise, num_paths_per_length] = ... % find_all_graph_paths(adjMtx, 'default', 1, sources, sinks);
    c = sum(sum(num_paths_pairwise));
end

% Get source nodes of a directed graph % Original version: Or Zuk, Hebrew University of Jerusalem
function s = get_graph_sources(A)
    s = find(sum(A - diag(diag(A)))==0);
end

% Get source nodes of a directed graph % Original version: Or Zuk, Hebrew University of Jerusalem
function s = get_graph_sinks(A)
    s = find(sum(A - diag(diag(A)),2)==0);
end

% Find all directed paths between any couple of vertices % Original version: Or Zuk, Hebrew University of Jerusalem % Edited by Ling Zhu & B. Pentland Michigan State University %
% Input:
% A - graph adjancy matrix
% output_format - whether to output indices of vertices seperated by length % (default), or binary vectors and then they're combined by length (BINARY)
% full_path_flag - 0: take all paths (including partial, default). 1: just from source to sink
%
% Output:
% paths - structure of all paths. Can be either binary or indices
% num_paths_pairwise - how many paths connect each pair of nodes
% num_paths_per_length - how many paths are there of each length

function [paths num_paths_pairwise num_paths_per_length] = ...  
    find_all_graph_paths(A, output_format, full_path_flag, source_nodes, sink_nodes)

if(~exist('output_format', 'var') || isempty(output_format))
    output_format = 'indices';
end
if(~exist('full_path_flag', 'var') || isempty(full_path_flag)) % default is to take all partial paths
    full_path_flag = 0;
end
N = 2 * length(A);
disp(strcat('** Limiting to paths of length = ', N));
% N = 10

tic:
disp('Step 1');
paths = cell(N,1); % paths[i] contains all paths of length i
for i=1:length(A) % was i=1:N % initialize paths
    paths{i} = cell(N);
    for j=1:length(A) % was j=1:N
        if(A(i,j))
            paths{1}{i,j} = [i j]; % zeros(1,N); paths{1}{i,j}(i) = 1;  paths{1}{i,j}(j) = 1;
        end
    end
end

disp('Step 2');
init_paths = paths{1};
for i=2:N % loop up to N
    tic;
    disp(strcat('Searching length of ', int2str(i), ' path')):
    paths{i} = path_matrix_mult(paths{i-1}, init_paths);
    toc;
end

disp('Step 3');
if(full_path_flag) % take only paths from source to sink
    for i=1:N % loop on path length
        for j=1:N
...
for k=1:N
    if(~any(j == source_nodes) || ~any(k == sink_nodes))
        paths{i}{j,k} = [];
    end
end
end
end
disp('Step 4');
num_paths_per_length = zeros(N,1);
num_paths_pairwise = zeros(N);
for i=1:N
    cur_num_paths = cellfun('size', paths{i}, 1); % 'uniformoutput', false);
    num_paths_per_length(i) = sum(cur_num_paths(:));
    num_paths_pairwise = num_paths_pairwise + cur_num_paths;
end
disp('Step 5');
switch output_format
    case {'binary', 'BINARY'}
        new_paths = cell(N);
        for i=1:N % unite paths of different lengths
            %                paths{1} = union_cell(paths{1}, paths{i}, 'rows');
            for j=1:N
                for k=1:N
                    tmp_paths = zeros(size(paths{i}{j,k}, 1), N);
                    for r = 1:size(paths{i}{j,k}, 1)
                        tmp_paths(r,paths{i}{j,k}(r,:)) = 1;
                    end
                    new_paths{j,k} = [new_paths{j,k}' tmp_paths']';
                end
            end
        end % unite different lengths paths
        paths = new_paths;
    end
toc;
return;

%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
% get a list of paths from two lists of paths
function C = path_matrix_mult(A, B)
N = length(A); C = cell(N);
for i=1:N
    for j=1:N
        C[i,j] = [];
    end
end
for k=1:N
    if(~any(j == source_nodes) || ~any(k == sink_nodes))
        paths{i}{j,k} = [];
    end
end
end
end
disp('Step 4');
num_paths_per_length = zeros(N,1);
num_paths_pairwise = zeros(N);
for i=1:N
    cur_num_paths = cellfun('size', paths{i}, 1); % 'uniformoutput', false);
    num_paths_per_length(i) = sum(cur_num_paths(:));
    num_paths_pairwise = num_paths_pairwise + cur_num_paths;
end
disp('Step 5');
switch output_format
    case {'binary', 'BINARY'}
        new_paths = cell(N);
        for i=1:N % unite paths of different lengths
            %                paths{1} = union_cell(paths{1}, paths{i}, 'rows');
            for j=1:N
                for k=1:N
                    tmp_paths = zeros(size(paths{i}{j,k}, 1), N);
                    for r = 1:size(paths{i}{j,k}, 1)
                        tmp_paths(r,paths{i}{j,k}(r,:)) = 1;
                    end
                    new_paths{j,k} = [new_paths{j,k}' tmp_paths']';
                end
            end
        end % unite different lengths paths
        paths = new_paths;
    end
toc;
return;
for k=1:N
    new_path = combine_paths(A{i,k}, B{k,j});
    % reduce the cycle during search paths - Ling
    if isempty(new_path),
        continue;
    end
    good_index=[];
    parfor search_i=1:size(new_path,1),
        if hascycle(new_path(search_i,:))==0
            good_index=[good_index;search_i];
        end
    end
    % disp(good_index);
    if isempty(good_index)~=1,
        new_path= new_path(good_index,:); %
    else
        new_path=[];
    end
    if(~isempty(new_path))
        C{i,j} = [C{i,j}' new_path'']; % unique_cell([C{i,j}, new_path])
    end
end
end
end

function C = combine_paths(A, B)

if isempty(A) || isempty(B)
    C = [];
    return;
end
% Old version (keeps binary vectors - faster but doesn't save directions)
n1 = size(A, 1); n2 = size(B, 1);
% C = max(repmat(A, n2, 1), repmat(B, n1, 1));
C = [repmat(A(:,1:end-1), n2, 1) repmat(B, n1, 1)]; % new version: no need for cell array
(all paths of same size)

%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
%% Added by Ling Zhu: hascycle(): C=1 if vector A has cycle, otherwise C=0;
function C = hascycle(A)
C=0;
% establish suffixes
suffix=[];
for i=1:length(A)-1,
if i~=1,
suffix=[suffix; A(i:length(A)).zeros(1,i-1)];
else
    suffix=[suffix; A(i:length(A))];
end
end

%disp(suffix);
%sort suffix strings
suffix = sortrows(suffix);
%find longest common prefix between two consecutive strings
%if the common prefix is longer than 2, then return true
for i=1:size(suffix,1),
    temp_suf = suffix(i,:);
    temp_suf = temp_suf(temp_suf~=0);
    common_index = find(suffix(:,1)==temp_suf(1));
    if length(common_index)==1,
        continue;
    end
    if length(find(suffix(common_index,2)==temp_suf(2))) >1,
        C=1;
        return;
    end
end

Appendix C: Estimated complexity, low density networks (d < 0.2)

function total_ties=simple_estimated_complexity(nodes, branches, loops)
% Original version: B. Pentland, Michigan State University
% It is best to to replace the coefficients below with ones that fit
% your data.

% nodes = number of nodes in the network
% branches = number of edges above the diagonal
% loops = number of edges below the diagonal

% The coefficients in this equation were estimated by fitting this
% equation to simulated data, with 5 <= N nodes <= 20, and a probability
% of branches or loops ranging from 0 < p(edge) < 0.3. In the simulated
% data, branches and loops vary independently, and the total network density
% ranges between 0 <= density <=0.3. R-squared >=0.95 consistently on
% several trials.
total_ties = exp(2.5988 - 0.30909*nodes + 0.42036*branches + 0.23811*loops - 0.0026285*branches^2 - 0.003174*loops^2);

end

Appendix D: Estimated complexity, high density networks (d > 0.2)

function c=estimated_complexity(n,d)
% Original version: B. Pentland, Michigan State University
% Based on Roberts, B., & Kroese, D. P. (2007). Estimating the Number of
% % n is the size of the network (number of nodes)
% d is the density of the network
% % First get the exponent
% delta = n-1+(3.32/n) - (5.16/(d*n));
% %now get the factor K
% SumK= 0;
% for k = 0:n-2
% SumK = SumK + factorial(n-2)/factorial(k);
% end
% this is the estimated complexity
% c = SumK * d.^delta:
end