Supplementary Information for Making Communities Show Respect for Order

V. Vasiliauskaite[1], T.S. Evans

Centre for Complexity Science, and Theoretical Physics Group,
Imperial College London, SW7 2AZ, U.K.

2nd February 2020

Appendix

A Siblinarity antichain partition

We require a function which measures the quality of our partition of the set of nodes into our "siblinarity antichains". There are two main aspects to such a function: imposing the antichain constraint and using defining node similarity using neighbourhood overlap.

Consider a directed graph (digraph) $\mathcal{G} = (\mathcal{V}, \mathcal{E})$ where $\mathcal{V}$ is the set of nodes and $\mathcal{E}$ is the set of edges, denoted (n, m) for an edge from node n to node m. Note we do not assume we have a DAG in what follows and we shall comment on this further at the end of this section.

Nodes in an antichain satisfy the condition that they are not weakly connected, that is there is no directed path between the two nodes in either direction. A directed path from n to m is a sequence of nodes in which consecutive nodes are linked by an edge in the correct direction [1]. That is $\{n_j | j \in \{0, 1, \ldots, \ell\}, n_0 = n, n_\ell = m, (n_j, n_{j+1}) \in \mathcal{E} \text{ for } j < \ell\}$. If there is a directed path in our graph between two nodes $n, m \in \mathcal{V}$ in *either* direction i.e. the two nodes are WEAKLY CONNECTED, we will denote this as $n \sim m$. An ANTICHAIN $\mathcal{A}$ is a subset of nodes which are not weakly connected to any of the other nodes in the same antichain, that is if $n, m \in \mathcal{A}$ then $n \not\sim m$.

The second aspect is a similarity measure for two nodes, that is $\mathrm{sim}(n, m)$ is a function which increases as the nodes n and m become more similar. Our aim is to use only the information encoded in the network, information which is always available. There are still many options but in our work here we will use the number of common neighbours. That is if $\mathcal{N}(n)$ is the number of neighbours of node n then we use

$$\mathrm{sim}(n, m) = |\mathcal{N}(n) \cap \mathcal{N}(m)| \,. \tag{1}$$

In a directed graph such as our DAGs there are three natural sets of neighbours we can define. We can use the PREDECESSORS OF n, denoted $\mathcal{N}^{(\mathrm{pre})}(n)$, that is the set of nodes with outgoing edges that end at n. Alternatively, we can use the SUCCESSORS OF NODE n, denoted $\mathcal{N}^{(\mathrm{suc})}(n)$, the set of nodes connected to with incoming edges that start from n. That is

$$\mathcal{N}^{(\mathrm{pre})}(n) = \{m | (m, n) \in \mathcal{E}\} \,, \qquad \mathcal{N}^{(\mathrm{suc})}(n) = \{m | (n, m) \in \mathcal{E}\} \,, \tag{2}$$

and as illustrated in Fig. A1. Finally we can also use both sets at the same time and use $\mathcal{N}^{(\mathrm{both})}(n) = \mathcal{N}^{(\mathrm{pre})}(n) \cup \mathcal{N}^{(\mathrm{suc})}(n)$ as our neighbours set.

We then need to say if a particular value for the similarity of two nodes is large or small. To do this we define a null model, typically a randomised version of our original network, and

[1]Corresponding Author.

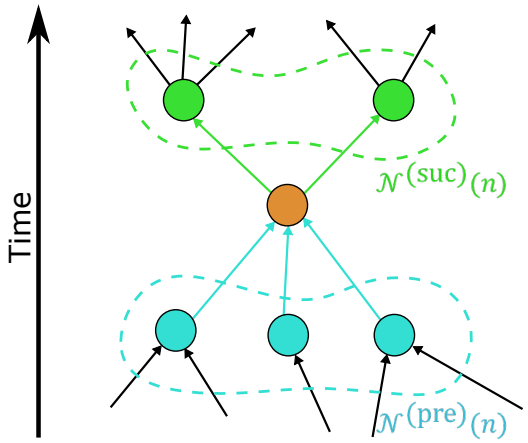


Figure A1: A figure to illustrate our neighbour set definitions.

we use the expected value in this null model for the similarity of two nodes n and m, which we will denote as $\mathrm{sim}_{\mathrm{null}}$.

Now we can put these elements together to define a function S that measures the quality of a given partition of our network into antichains, denoted as the set $\mathfrak{A}$. We consider a partition to be good if our antichains contain similar nodes. For instance in a family tree, we might want to group the biological siblings of mother and father pair. There is no direct biological connection between the siblings but they all have the same mother and father in common so the overlap in their precursor neighbour set, $\mathcal{N}^{(\mathrm{pre})}$ in the example shown in Fig. A2. We are

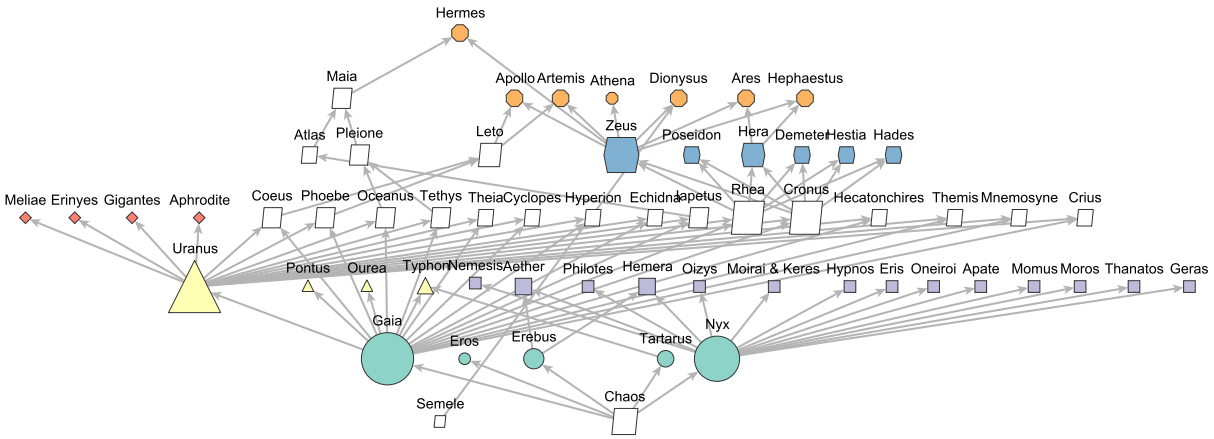


Figure A2: A family tree of Greek gods based on data from Wikipedia [2] (see section E.1). Links are from parents to offspring. Colours of nodes and their shapes both show grouping of deities using siblinarity based on common predecessors. For a comparison see the $\lambda = 2$ example in Fig. A3 which uses both successor and predecessor neighbours. White vertices indicate nodes in an antichain community of size one. The size of a node indicates its total degree.

aiming to find a partition of our set of nodes into antichains which we refer to as an ANTICHAIN PARTITION and which we denote as $\mathfrak{A}$. That is each element of the partition $\mathcal{A} \in \mathfrak{A}$ is an antichain.

Motivated by this family tree example, we call our quality function SIBLINARITY and we

denote this $S(\mathfrak{A})$ for a given antichain partition $\mathfrak{A}$. The generic form we choose is

$$S(\mathfrak{A}) = \sum_{\mathcal{A}\in\mathfrak{A}} \sum_{n\in\mathcal{A}} \sum_{m\in\mathcal{A}\setminus n} \left(\mathrm{sim}(n,m) - \mathrm{sim}_{\mathrm{null}}(n,m)\right) . \tag{3}$$

Note that there is no contribution from $n = m$ in this expression. This leads to the result that $S(\mathfrak{A})$ is zero for the trivial antichain partition, the one where each nodes is in an antichain by itself, i.e. $\mathfrak{A}_{\mathrm{trivial}}$ where

$$\mathfrak{A}_{\mathrm{trivial}} = \{\{v\}|v \in \mathcal{V}\} . \tag{4}$$

As noted there are many possible choices for the similarity function and the null model used for comparison. In practice what we use here is given by (1) which gives us

$$S(\mathfrak{A}) = \sum_{\mathcal{A}\in\mathfrak{A}} \sum_{n\in\mathcal{A}} \sum_{m\in\mathcal{A}\setminus n} \left(|\mathcal{N}(n)\cap\mathcal{N}(m)| - \mathbb{E}(|\mathcal{N}(n)\cap\mathcal{N}(m)|)\right) . \tag{5}$$

The outer sum is over all antichains $\mathcal{A}$ in the antichain partition; the inner sum is over all pairs of nodes in a given antichain $\mathcal{A}$. A contribution to the total siblinarity from a pair of nodes n and m is equal to the size of the intersection between their neighbours (predecessors or successors or perhaps both) minus the expected value of the size of this intersection, $\mathbb{E}(|\mathcal{N}(n)\cap\mathcal{N}(m)|)$. The expected value depends on the choice of the null model.

For instance, we use a configuration model [1] as a simple null model in which the DAG has been randomised maintaining the degree of every node and the directions of the edge but otherwise the order of the original DAG has been lost. Consider one term in our expression and so focus on a given pair of nodes n and m in the same antichain $\mathcal{A}$. Then pick one of the $|\mathcal{N}(n)|$ neighbours of node n, say node p. For simplicity we imagine that we are looking at successors so that this neighbour p is at the end of an edge leaving n. In our simple configuration null model this neighbouring node will have in-degree $\langle(k^{(\mathrm{in})})^2\rangle/\langle k^{(\mathrm{in})}\rangle$. That is neighbouring nodes of node n have on average $(\langle(k^{(\mathrm{in})})^2\rangle - 1)k_n^{\mathrm{out}}/\langle k^{(\mathrm{in})}\rangle$ incoming edges which could be at the end of edges from node m. Given node m has k_m^{out} edges, the number of common neighbours may be estimated to be

$$\mathbb{E}(|\mathcal{N}(n)\cap\mathcal{N}(m)|) \approx \frac{(\langle(k^{(\mathrm{in})})^2\rangle - 1)k_n^{\mathrm{out}}k_m^{\mathrm{out}}}{\langle k^{(\mathrm{in})}\rangle|\mathcal{E}|} . \tag{6}$$

While this could be tried as a null model in our siblinarity expressions, we chose not to do so. Rather we first rewrite our siblinarity in terms of matrices and then use that representation to inspire our choice of null model.

We can rewrite (5) in terms of the adjacency matrix $\mathbf{A}$ for our DAG [3]. We use the convention that $\mathbf{A}_{nm}$ is the weight of the edge from n to m, with zero weight for no edge. Consider $\tilde{\mathbf{A}}$ as an effective similarity matrix obtained from the product of the adjacency matrix and its transpose. In the case where we have an unweighted DAG, $\tilde{\mathbf{A}}^{(\mathrm{suc})} = \mathbf{A}.\mathbf{A}^{\mathrm{T}}$ is our successors-based similarity matrix whereas $\tilde{\mathbf{A}}^{(\mathrm{pre})} = \mathbf{A}^{\mathrm{T}}.\mathbf{A}$ is a similarity matrix based on predecessors, so emulating the expressions in (2). Should we choose to use both sets of neighbours then we simply use the sum of these two matrices $\tilde{\mathbf{A}}^{(\mathrm{both})} = \tilde{\mathbf{A}}^{(\mathrm{suc})} + \tilde{\mathbf{A}}^{(\mathrm{pre})}$. Whichever of these effective similarity matrices $\tilde{\mathbf{A}}$ we use, it means we may write our siblinarity function in the following form:

$$S(\mathfrak{A}) = \sum_{n} \sum_{m|m\neq n} \left(\tilde{A}_{nm} - \frac{\kappa_n\kappa_m}{W}\right)\delta(\mathcal{A}_n,\mathcal{A}_m) , \quad \text{where } n\in\mathcal{A}_n\in\mathfrak{A} , \; m\in\mathcal{A}_m\in\mathfrak{A} . \tag{7}$$

Here the $\delta(\mathcal{A}_n, \mathcal{A}_m)$ is one if n and m are in the same antichain ($\mathcal{A}_n = \mathcal{A}_m$), zero if they are in different antichains ($\mathcal{A}_n \cap \mathcal{A}_m = \emptyset$). We define κ_n to be the effective strength of edges attached to a node n in the similarity matrix $\tilde{\mathbf{A}}$ so $\kappa_n = \sum_m \tilde{A}_{nm}$, while and $W = \sum_{n,m} \tilde{A}_{nm}$ is the total weight of edges in the similarity matrix.

Note that the form of (7) means that we have chosen an explicit form for our null model. This matrix form (7) has been chosen to emulate the modularity function [4] used as a measure of the quality of a partition of nodes in a weighted undirected network with adjacency matrix $\tilde{\mathbf{A}}$. The null model we use is one in which we look at a "second-neighbour network" whose adjacency matrix is $\tilde{\mathbf{A}}$. This has the same nodes as the original DAG $\mathcal{G}$ with undirected but weighted edges present between nodes if the are second neighbours in the original DAG. These second neighbours in the original DAG $\mathcal{G}$ are defined by going one step forwards and one step backwards if we are using successor neighbourhoods, and similarly for other choice of neighbourhood $\mathcal{N}$ of (2). The null model is a configuration model [1] for this second-neighbour network.

Also note that the form given here includes non-zero diagonal entries $\tilde{A}_{nn}$ with corresponding contributions to the strength's κ_n. For instance if the original DAG $\mathcal{G}$ is unweighted then $\tilde{A}_{nn}$ is the number of (first) neighbours of node n in the DAG $\mathcal{G}$. One could eliminate the self-loops from the second-neighbour network producing a non-backtracking form for the adjacency matrix say $\tilde{\mathbf{A}}^{(\mathrm{NBT})}$ instead of $\tilde{\mathbf{A}}$. For instance we could use $\tilde{A}^{(\mathrm{NBT,suc})}_{mn} = (\tilde{A}^{(\mathrm{suc})})_{mn} - k^{\mathrm{out}}_n \delta_{mn}$ instead of $\tilde{\mathbf{A}}^{(\mathrm{suc})}$. We see no strong reason to use this non-backtracking form and have not considered $\tilde{\mathbf{A}}^{(\mathrm{NBT})}$ here. Equally, apart from algebraic simplicity, we can no reason not to use the non-backtracking form $\tilde{\mathbf{A}}^{(\mathrm{NBT})}$ but choose not to pursue this further here.

The big difference between siblinarity and modularity is that for our context, our partitions are restricted to be antichains, something implicit in our $\mathfrak{A}$ notation. Apart from this important restriction, we are working on the modularity of a derived weighted but undirected network with adjacency matrix given by $\tilde{\mathbf{A}}$. If our original DAG was unweighted, this effective adjacency matrix counts the number of 'routes' (not a path in the usual precise definition used in graph theory) which consist of one forward and one backwards step on our original DAG.

A.1 Resolution

It is worth noting that we can control the resolution of obtained partition by scaling the null model contribution in the siblinarity function (3) by a parameter λ. This mimics one way that the resolution can be changed for community detection using modularity [5]. In our case we suggest a modified form for the generic siblinarity function

$$S(\mathfrak{A}, \lambda) = \sum_{\mathcal{A} \in \mathfrak{A}} \sum_{n \in \mathcal{A}} \sum_{m \in \mathcal{A} \setminus n} \left(\mathrm{sim}(n, m) - \lambda \, \mathrm{sim}_{\mathrm{null}}(n, m) \right) . \tag{8}$$

This clearly reduces to the original equation, (5), when $\lambda = 1$. Large values of λ would yield smaller antichains as adding nodes to an antichain produces a penalty. So for large enough λ, the antichain partition which maximises the modified siblinarity will be where each antichain contains just one node. When λ is zero, any two nodes which are not connected by a path but share at least one shared neighbour will increase the siblinarity value if put in the same antichain. So for small λ we expect the maximal modified siblinarity is likely to be something that has the fewest number of antichains. In particular, a negative λ will allow nodes with no path between them and with no common neighbours to to have larger siblinarity values if they are in the same antichain rather than each node being in an antichain of one node.

To illustrate this idea, we find antichain communities by maximising a modified form of our usual matrix siblinarity (7), namely

$$S(\mathfrak{A},\lambda) \;=\; \sum_{\mathcal{A}\in\mathfrak{A}} \sum_{n\in\mathcal{A}} \sum_{m\in\mathcal{A}\setminus n} \big(\tilde{A}_{nm} - \lambda\frac{\kappa_n\kappa_m}{W}\big)\,. \tag{9}$$

We use $S(\mathfrak{A},\lambda)$ to find antichain communities in the family tree of the Greek gods [2] (see section E.1) shown in Fig. A3. As expected, when λ is small, such as in A, communities are large and very close to height antichains. Larger values of λ in B and C produce more refined partitions. The same behaviour can be sen in a very simple example in section B.

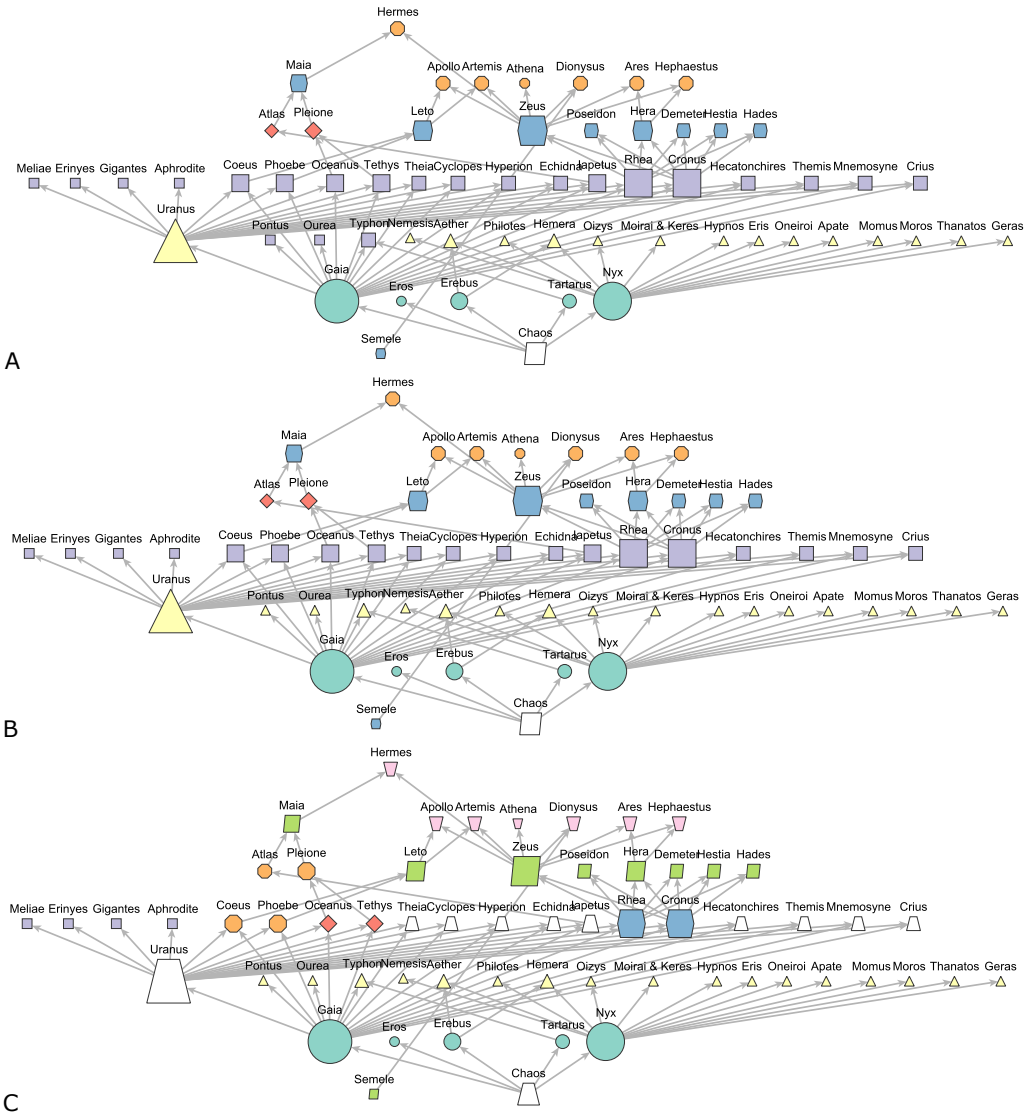
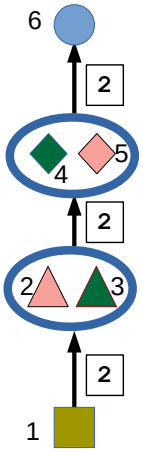


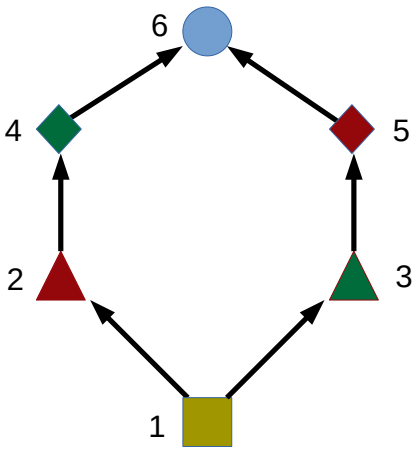
Figure A3: Antichains in the family tree of Greek Gods [2] (see section E.1), obtained by maximising the modified siblinarity $S(\mathfrak{A}, \lambda)$ (9) using different resolution parameters λ: in A $\lambda = 0.5$, in B $\lambda = 1.5$, in C $\lambda = 2$. Different colours an shapes show different siblinarity communities, obtained by considering both, future and past neighbours of nodes. As expected, when λ is small, communities are large and very close to height antichains. Larger values of λ produce more refined partitions. White vertices indicate nodes in an antichain community of size one. Note that we can compare this $\lambda = 2$ example against the antichain communities in Fig. A2 as the latter uses predecessors unlike here where both successor and predecessor neighbours are used to evaluate siblinarity.

B A Simple example

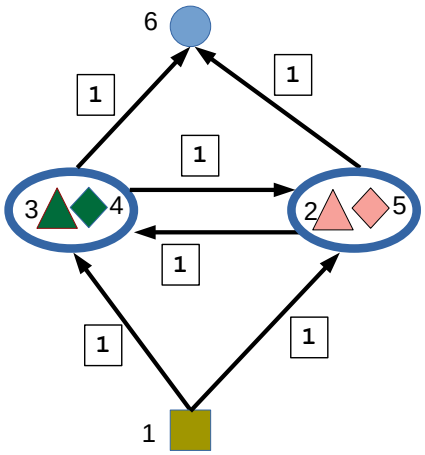
Consider the network G shown in the centre of Fig. B4. Numbering the vertices from the



$\mathcal{H}$ for $\mathfrak{A}_2$
$\{\{1\},\ \{2,3\},\ \{4,5\},\ \{6\}\}$.



$\mathcal{G}$



$\mathcal{H}$ for $\mathfrak{A}_3$
$\{\{1\},\ \{2,5\},\ \{3,4\},\ \{6\}\}$.

Figure B4: In the centre the simple example G considered in the text. On the left is the reduced network $\tilde{G}_2$ based on the height antichain partition $\mathfrak{A}_2$ of (15). On the right is the reduced network $\tilde{G}_3$ based on the antichain partition $\mathfrak{A}_3$ of (17). Note that this last antichain partition $\mathfrak{A}_3$ produces a reduced network which is a directed graph with cycles. In the networks on the left and right, the numbers in squares indicate the weight of the nearest edge. The central network is unweighted. For the reduced networks on the left and right, the ovals are the nodes in the centre but within they show which vertices were merged to form the new node in the reduced graph.

bottom up we have edges $(1,2)$, $(1,3)$, $(2,4)$, $(3,5)$, $(4,6)$, and $(5,6)$ where (m,n) is an edge from vertex m to vertex n. Using a convention that A_{mn} is an edge from m to n we have that columns are labelled by n and rows are labelled by m. So G in Fig. B4 has adjacency matrix

$$\mathbf{A} = \begin{pmatrix} 0 & 1 & 1 & 0 & 0 & 0 \\ 0 & 0 & 0 & 1 & 0 & 0 \\ 0 & 0 & 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & 0 & 0 & 1 \\ 0 & 0 & 0 & 0 & 0 & 1 \\ 0 & 0 & 0 & 0 & 0 & 0 \end{pmatrix} \tag{10}$$

In this example we will also use the following definition of siblinarity

$$\hat{S}(\mathfrak{A},\lambda) = \sum_{\mathcal{A}\in\mathfrak{A}}\sum_{n\in\mathcal{A}}\sum_{m\in\mathcal{A}}\left(\tilde{A}_{nm} - \lambda\frac{\kappa_n\kappa_m}{W}\right), \tag{11}$$
$$\kappa_n := \sum_m \tilde{A}_{nm}\,, \quad W = \sum_{n,m}\tilde{A}_{nm}\,.$$

Note that we are not excluding $m = n$ in this definition of siblinarity $\hat{S}(\mathfrak{A},\lambda)$ (with $\hat{S}(\mathfrak{A}) = \hat{S}(\mathfrak{A},\lambda=1)$) unlike the definition of $S(\mathfrak{A})$ of (9). The difference is a constant independent of

$\mathfrak{A}$. Since $S(\mathfrak{A})$ of (11) is zero for the trivial antichain partition $\mathfrak{A}_{\text{trivial}}$ of (4), in which every node is placed in their own element of the partition, this difference is equal to the value of $\hat{S}(\mathfrak{A})$ for that trivial antichain partition. In our case this is $\hat{S}(\mathfrak{A}_1)$ below. We will choose to use a successor form of the two-step matrix in (11), that is $\tilde{\mathbf{A}}^{(\text{suc})} = \mathbf{A}.\mathbf{A}^{\text{T}}$, so that

$$\tilde{\mathbf{A}}^{(\text{suc})} = \begin{pmatrix} 2 & 0 & 0 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 & 0 & 0 \\ 0 & 0 & 1 & 0 & 0 & 0 \\ 0 & 0 & 0 & 1 & 1 & 0 \\ 0 & 0 & 0 & 1 & 1 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 \end{pmatrix} \tag{12}$$

Each vertex forms on antichain

Now suppose we use this for a partition where each element of the partition is a single vertex. This is always an antichain partition. Here we have

$$\mathfrak{A}_1 = \{\{1\},\ \{2\},\ \{3\},\ \{4\},\ \{5\},\ \{6\}\} \tag{13}$$

and we find that the siblinarity value is

$$\begin{aligned} \hat{S}(\mathfrak{A}_1, \lambda) &= \left(2 - \lambda\frac{2.2}{8}\right) + \left(1 - \lambda\frac{1.1}{8}\right) + \left(1 - \lambda\frac{1.1}{8}\right) + \left(1 - \lambda\frac{2.2}{8}\right) + \left(1 - \lambda\frac{2.2}{8}\right) + 0 \\ &= 6 - \lambda\frac{14}{8}\,. \end{aligned} \tag{14}$$

The Height Antichain Partition

The height antichain partition (same as the depth antichain partition here) is

$$\mathfrak{A}_2 = \{\{1\},\ \{2,3\},\ \{4,5\},\ \{6\}\}\ . \tag{15}$$

This has siblinarity equal to

$$\hat{S}(\mathfrak{A}_2, \lambda) = \left(2 - \lambda\frac{2.2}{8}\right) + \left(2 - \lambda 3.\frac{1.1}{8}\right) + \left(4 - \lambda 3.\frac{2.2}{8}\right) + 0 = 6 - \lambda\frac{19}{8}\,. \tag{16}$$

Antichain Partition with Cyclic Derived Graph

For an antichain partition of

$$\mathfrak{A}_3 = \{\{1\},\ \{2,5\},\ \{3,4\},\ \{6\}\} \tag{17}$$

the derived graph $\tilde{G}$ will be directed but not cyclic as shown in Fig. B4. The siblinarity for this antichain partition is

$$\begin{aligned} \hat{S}(\mathfrak{A}_3, \lambda) &= \left(2 - \lambda\frac{2.2}{8}\right) + \left(2 - \lambda[\frac{1.1}{8} + \frac{1.2}{8} + \frac{2.2}{8}]\right) + \left(2 - \lambda[\frac{1.1}{8} + \frac{1.2}{8} + \frac{2.2}{8}]\right) + 0 \\ &= 6 - \lambda\frac{18}{8}\,. \end{aligned} \tag{18}$$

Antichain Partition Four

Another possible antichain partition is

$$\mathfrak{A}_4 = \{\{1\},\ \{2\},\ \{3\},\ \{4,5\},\ \{6\}\} \tag{19}$$

This gives

$$\hat{S}(\mathfrak{A}_4, \lambda) = \left(2 - \lambda\frac{2.2}{8}\right) + \left(1 - \lambda\frac{1.1}{8}\right) + \left(1 - \lambda\frac{1.1}{8}\right) + \left(3 - \lambda 3.\frac{2.2}{8}\right) + 0 = 7 - \lambda\frac{18}{8}\,. \tag{20}$$

Antichain Partition Five

The fifth antichain partition we consider is

$$\mathfrak{A}_5 = \{\{1\},\ \{2,3\},\ \{4\},\ \{5\},\ \{6\}\} \tag{21}$$

This is similar to antichain four but it is not related by any symmetry so gives a different result.

$$\hat{S}(\mathfrak{A}_5, \lambda) = \left(2 - \lambda\frac{2.2}{8}\right) + \left(2 - \lambda 3.\frac{1.1}{8}\right) + \left(1 - \lambda\frac{2.2}{8}\right) + \left(1 - \lambda\frac{2.2}{8}\right) + 0 = 6 - \lambda\frac{15}{8}\,. \tag{22}$$

Antichain Partition Six and Seven

The last two possible antichain partitions have the same siblinarity by symmetry

$$\mathfrak{A}_6 = \{\{1\},\ \{2,5\},\ \{3\},\ \{4\},\ \{6\}\} \tag{23}$$

and

$$\mathfrak{A}_7 = \{\{1\},\ \{2\},\ \{3,4\},\ \{5\},\ \{6\}\} \tag{24}$$

These have siblinarity values of

$$\begin{aligned}\hat{S}(\mathfrak{A}_6, \lambda) = \hat{S}(\mathfrak{A}_7, \lambda) &= \left(2 - \lambda\frac{2.2}{8}\right) + \left(1 - \lambda\frac{1.1}{8}\right) + \left(2 - \lambda[\frac{1.1}{8} + \frac{1.2}{8} + \frac{2.2}{8}]\right) \\ &\quad + \left(1 - \lambda\frac{2.2}{8}\right) + 0 \\ &= 6 - \lambda\frac{16}{8}\,.\end{aligned} \tag{25}$$

Siblinarity Maximisation

The results for all the possible antichains for the DAG G of Fig. B4 are summarised in Table B1. The best partition for maximum siblinarity is not in fact the height partition $\mathfrak{A}_2$, (this is second best) but is the fourth partition $\mathfrak{A}_4$. This because the two antichains $\{2\}$ and $\{3\}$ are preferred to the single antichain $\{2,3\}$ and this is because nodes 2 and 3 have no common successors. Had we used a siblinarity that involved predecessors in some way then $\{2,3\}$ would have been preferred.

When we introduce the resolution parameter λ in (9), the modified siblinarity gives the height antichain community $\mathfrak{A}_2$ (15), a maximal antichain partition, as the best solution for $\lambda < 0$. For $0 < \lambda < 2$ we find the fourth antichain $\mathfrak{A}_4$ of (19) is the optimal. The trivial partition with each antichain containing one node, $\mathfrak{A}_1$ of (13), has the largest modified siblinarity for $\lambda > 2$.

Antichain Partition		Siblinarity		Mod.Siblinarity	
		$\hat{S}(\mathfrak{A})$	$S(\mathfrak{A})$	$\hat{S}(\mathfrak{A},\lambda)$	$S(\mathfrak{A},\lambda)$
$\mathfrak{A}_1$	$\{\{1\},\{2\},\{3\},\{4\},\{5\},\{6\}\}$	$\frac{34}{8}$	$\frac{0}{8}$	$6-\lambda\frac{14}{8}$	0
$\mathfrak{A}_2$	$\{\{1\},\{2,3\},\{4,5\},\{6\}\}$	$\frac{37}{8}$	$+\frac{3}{8}$	$7-\lambda\frac{19}{8}$	$1-\lambda\frac{5}{8}$
$\mathfrak{A}_3$	$\{\{1\},\{2,5\},\{3,4\},\{6\}\}$	$\frac{30}{8}$	$-\frac{4}{8}$	$6-\lambda\frac{18}{8}$	$-\lambda\frac{4}{8}$
$\mathfrak{A}_4$	$\{\{1\},\{2\},\{3\},\{4,5\},\{6\}\}$	$\frac{38}{8}$	$+\frac{4}{8}$	$7-\lambda\frac{18}{8}$	$1-\lambda\frac{4}{8}$
$\mathfrak{A}_5$	$\{\{1\},\{2,3\},\{4\},\{5\},\{6\}\}$	$\frac{33}{8}$	$-\frac{1}{8}$	$6-\lambda\frac{15}{8}$	$-\lambda\frac{1}{8}$
$\mathfrak{A}_6$	$\{\{1\},\{2,5\},\{3\},\{4\},\{6\}\}$	$\frac{32}{8}$	$-\frac{2}{8}$	$6-\lambda\frac{16}{8}$	$-\lambda\frac{2}{8}$
$\mathfrak{A}_7$	$\{\{1\},\{2\},\{3,4\},\{5\},\{6\}\}$	$\frac{32}{8}$	$-\frac{4}{8}$	$6-\lambda\frac{16}{8}$	$-\lambda\frac{2}{8}$

Table B1: A table of siblinarity values for the different antichain partitions of the graph G of Fig. B4. We are using the modified versions of siblinarity with a resolution parameter λ, that is $S(\mathfrak{A},\lambda)$ of (9) and $\hat{S}(\mathfrak{A},\lambda)$ of (11). The $\lambda=1$ values are also given as $S(\mathfrak{A})=S(\mathfrak{A},\lambda=1)$ and $\hat{S}(\mathfrak{A})=\hat{S}(\mathfrak{A},\lambda=1)$. Note that the values of siblinarity $S(\mathfrak{A},\lambda)$ and $\hat{S}(\mathfrak{A},\lambda)$ are related through the value for the largest (trivial) partition $\mathfrak{A}_1$ since $S(\mathfrak{A},\lambda)=\hat{S}(\mathfrak{A},\lambda)-\hat{S}(\mathfrak{A}_1,\lambda)$.

C Louvain Siblinarity Optimisation

Having defined a quality function, we can look for a partition of our nodes into antichains, $\mathfrak{A}$, which maximises the siblinarity $S(\mathfrak{A})$. This task faces the same challenges as most network community detection methods; there are many local minima and only approximate solutions can be found in a reasonable amount of computational time. Here, we will discuss how to adapt the Louvain algorithm [6] which is a widely used and successful methods to find communities in networks which maximise modularity. Emulating the Louvain algorithm, our siblinarity maximisation method is an iterative greedy algorithm in which each iteration has two phases.

In the first phase of our algorithm, we start with an initial partition into antichains in which each node is assigned to its own antichain. At each subsequent step, we try to move a single node n from its current antichain, $\mathcal{A}_a$, to another antichain $\mathcal{A}_b$, always choosing the configuration which maximises the siblinarity, even if that means leaving the antichains unchanged. In our implementation, we visit each node in a fixed sequence. Once the sequence is exhausted, we sweep through the same sequence once again. This process is continued until there are no more changes in the optimal antichain partition possible whatever node n we choose to examine. That is when changing the antichain partition by moving just one node can not increase the siblinarity. This marks the end of the first phase. In principle, we can also stop this first phase at any point as every new antichain partition is, by definition better than the last. So in our algorithm we also stop this phase if we have completed a given number of sweeps since the second phase is almost certain to simplify the problem and so speed up subsequent iterations.

For each node n we choose for a possible move, the change in siblinarity is calculated for removing n from its current antichain, $\mathcal{A}_a$, and placing in a new antichain A_b. It is important to enforce the constraint that the node n must not be connected to any existing node m in the potential new antichain $\mathcal{A}_b$, i.e. we want $\mathcal{A}_b\cup\{n\}$ to be an antichain. In our algorithm, we further limit the choice of which new antichains $\mathcal{A}_b$ we examine. If we are using siblinarity

defined using a similarity measure using a neighbourhood set $\mathcal{N}(n)$ for our nodes n, then we look for antichains $\mathcal{A}_b$ which contain at least one node $m \in \mathcal{A}_b$ which has a non-trivial similarity measure with our chosen node n, i.e. for us we require $|\mathcal{N}(n) \cap \mathcal{N}(m)| > 0$. These potential new antichains for node n are easy to find as this involves a two-step walk on the network starting from n. If we use a neighbourhood based on successors, that is $\mathcal{N}^{(\mathrm{suc})}(n)$, then we only look at antichains $\mathcal{A}_b$ which contain a predecessor m of a successor of n. We will call these SUCCESSOR ANTICHAINS. If we look only at predecessors neighbourhoods and so use $\mathcal{N}^{(\mathrm{pre})}(n)$ for our neighbourhood sets, we shall refer to the resulting antichain partitions as PREDECESSOR ANTICHAINS. There is only one other case we examine, and that is we also check the case where we allow n to join a new antichain consisting of the node n alone, i.e. $\mathcal{A}_b = \emptyset$.

The change in siblinarity ΔS is given by

$$\begin{aligned}\Delta S(\mathcal{A}_a, \mathcal{A}_b \to \mathcal{A}_a \backslash n, \mathcal{A}_b \cup \{n\}) = & \sum_{m \in \mathcal{A}_b} \left(|\mathcal{N}(n) \cap \mathcal{N}(m)| - \mathbb{E}(|\mathcal{N}(n) \cap \mathcal{N}(m)|)\right) \\ & - \sum_{q \in \mathcal{A}_a \backslash n} \left(|\mathcal{N}(n) \cap \mathcal{N}(q)| - \mathbb{E}(|\mathcal{N}(n) \cap \mathcal{N}(q)|)\right) , \qquad (26) \\ & \text{provided } n \not\sim \mathcal{A}_b \, .\end{aligned}$$

The first term is the contribution from the addition of node n to the antichain $\mathcal{A}_b$, while the second term is the effect of the removal of node n from its current antichain $\mathcal{A}_a$. Note the condition that n is not connected to any node in the existing antichain $\mathcal{A}_b$ which we denote as $n \not\sim \mathcal{A}_b$. This is needed to ensure $\mathcal{A}_b$ remains an antichain when n is added. Computationally it requires a check if there are no paths in the network between any pair of nodes of an antichain. This can be done in several ways. For instance, an entry A'_{nm} of a matrix $\mathbf{A'} = \sum_{i=1}^{\ell} Amatr^i$, where ℓ is the length of the longest path in the network, is only zero if there is no path from n to m. So if $A'_{nm} + A'_{mn} = 0$, the nodes n, m can be in the same antichain.

In the matrix notation of (7), the change in siblinarity is given by

$$\begin{aligned}\Delta S(\mathcal{A}_a, \mathcal{A}_b \to \mathcal{A}_a \backslash n, \mathcal{A}_b \cup \{n\}) = & \sum_{m \in \mathcal{A}_b} \left(\tilde{A}_{nm} - \frac{\kappa_n \kappa_m}{W}\right) - \sum_{q \in \mathcal{A}_a \backslash n} \left(\tilde{A}_{nq} - \frac{\kappa_n \kappa_q}{W}\right) \qquad (27) \\ & \text{provided } n \not\sim \mathcal{A}_b \, .\end{aligned}$$

In the second phase we create an *induced graph* $\mathcal{H} = \{\mathcal{V}_{\mathrm{H}}, \mathcal{E}_{\mathrm{H}}\}$ from the original graph $\mathcal{G}$ and the antichain partition $\mathfrak{A}$ left at the end of phase one. Each node $a \in \mathcal{V}_{\mathrm{H}}$ in this induced graph $\mathcal{H}$ represents a single antichain, $\mathcal{A}_a \in \mathfrak{A}$, as given at the end of the previous phase. The edges between nodes of induced graph are given a weight equal to the sum of the weights of all the edges between the equivalent antichain nodes in the original graph $\mathcal{G}$ of the induced graph. For instance, if there were k_{ba} edges all of weight 1 pointing from nodes in the antichain $\mathcal{A}_a$ to the antichain $\mathcal{A}_b$ at the end of phase one, there would be a directed edge $(a, b) \in \mathcal{E}_{\mathrm{H}}$ in the induced graph with weight equal to k_{ba} in the induced graph[2]. In terms of matrices, if H_{ba} is equal to the weight of the edge from node a to b in the adjacency matrix for the induced graph, then we have that

$$H_{ba} = \sum_{m \in \mathcal{A}_a} \sum_{n \in \mathcal{A}_b} A_{nm} \, . \qquad (28)$$

[2]Induced graph does not have to be a DAG: antichains are possible in graphs with cycles as Fig. B4 shows. By definition, an antichain is a subset of nodes such that there is no path between any of two of them in this subset. This is perfectly valid in any graph, however, in some they are more interesting than in others.

Once the induced graph is created, the algorithm continues by applying finding an antichain partition of the induced graph using siblinarity, starting with the phase one. The algorithm continues until there is no substantial increase in the siblinarity function (5).

D Basic Statistics on Antichains

One way to look at antichains is to look at bipartite network $\mathcal{B}(\mathcal{A})$ associated with each antichain. For each antichain $\mathcal{A}$ of a graph $\mathcal{G} = (\mathcal{V}, \mathcal{E})$, we define a bipartite network $\mathcal{B}(\mathcal{A})$. The first type of vertex in the bipartite network are simply those in the antichain itself, $\mathcal{A}$. The second type of vertex in $\mathcal{B}(\mathcal{A})$ is the set of all neighbours of the antichain vertices, that is

$$\mathcal{N}(\mathcal{A}) = \cup_{v \in \mathcal{A}} \mathcal{N}^{(\text{suc})}(v)\,. \tag{29}$$

An edge of the bipartite graph $\mathcal{B}(\mathcal{A})$ is present if there is a corresponding connection between a given vertex in the antichain $\mathcal{A}$ to any of its neighbours, any vertex in $\mathcal{N}(\mathcal{A})$

$$\mathcal{E}(\mathcal{A}) = \{(v,n) | n \in \mathcal{N}(\mathcal{A}),\, v \in \mathcal{A},\, (v,n) \in \mathcal{E}\} \cup \{(n,v) | n \in \mathcal{N}(\mathcal{A}),\, v \in \mathcal{A},\, (v,n) \in \mathcal{E}\}\,. \tag{30}$$

In practice, we also remove loosely connected neighbour nodes but for simplicity we will not indicate that in our definitions here. This gives us $\mathcal{B}(\mathcal{A}) = (\mathcal{A} \cup \mathcal{N}(\mathcal{A}), \mathcal{E}(\mathcal{A}))$.

There are many possible network measurements we can make on each bipartite graph which can help us understand the nature of the antichains found in any example. In our work we focussed on some of the simplest measures.

- $|\mathcal{A}|$ — The size of an antichain, i.e. the number of nodes in an antichain.
- $|\mathcal{N}|$ — The number of neighbours of an antichain.
- $\langle k \rangle$ — The average number of neighbours of nodes in an antichain.

$$\langle k \rangle_{\mathcal{A}} = \frac{|\mathcal{E}(\mathcal{A})|}{|\mathcal{A}|} \tag{31}$$

- $\sigma(k)_{\mathcal{A}}$ — The standard deviation of the number of neighbours of nodes in an antichain.

$$\sigma(k)_{\mathcal{A}} = \sqrt{\frac{\sum_{n \in \mathcal{A}} (k_n - \langle k \rangle_{\mathcal{A}})}{|\mathcal{A}| - 1}} \tag{32}$$

- $\langle k \rangle / |\mathcal{N}| = \frac{|\mathcal{E}(\mathcal{A})|}{|\mathcal{A}||\mathcal{N}|}$ — This is the density of the bipartite graph, the number of edges divided by the maximum number possible.

The average degree and standard deviation of degree of nodes in the antichain, $\langle k \rangle$ and $\sigma(k)$, give us a picture of how the even the connections between antichain nodes and their neighbours are. If $\langle k \rangle \approx |\mathcal{N}|$ and $\sigma(k) \approx 0$, we know that nodes of very similar degrees are joined together in an antichain, and their neighbourhoods are largely overlapping.

The ratio between $\langle k \rangle$ and $|\mathcal{N}|$ tell us how similar our bipartite graph is to a sparse "zig-zag" pattern. If $\langle k \rangle$ is small in comparison to $\mathcal{N}$, then we can expect small overlap overall between all nodes in the antichain. This statistic approaches 1 if every node in the antichain is connected to every neighbour.

E Additional Examples of Siblinarity and Data

E.1 Greek Gods

We have named the function used to find antichain partitions "siblinarity" using the analogy with a family tree. We use the number of common neighbours to link nodes in an antichain just as siblings share biological parents. Since family trees based on biological parentage are predominantly trees, directed acyclic graphs but which have few if any loops in the undirected version, we have looked to fiction to provide a more interesting example of a family tree. We have taken information on Greek gods from Wikipedia [2] with each node representing a Greek god with edges from a god to a child. Fig. A2 shows the results of applying our siblinarity clustering siblinarity based on common predecessors to this data set. See [7] for the original data.

E.2 Python Dependencies

Many software programmes can be extended by adding packages, extra programmes which extend the functionality of the core package. As the number of packages grows, some of the added packages start to use the functionality of some of the other packages added to the original core application. In order to ensure that each package can run correctly, there is usually a system to noting the dependencies of each package, that is, what other packages are required in order to run any one package.

A good example of such an software 'ecosystem' is the computer language `python` [8]. The `PyPI` repository [9] of packages for python records over 180,000 projects at the time of writing. To illustrate the principle, we used a python `python` installation on one of the author's machines in 2019. We used V.Naik's `pipdeptree` [10] (itself a python package listed on `PyPI`) to extract the directed acyclic graph representing the package dependency DAG. Each node is a package and we add a link from the parent package to the sibling, where the sibling package is requires the installation of the parent in order to work. The results are shown in Fig. E5 and the data used is provide on [7].

This illustrates several of the points made elsewhere. For instance, the `scipy`, `pandas`, and `matplotlib` packages, bright red nodes in the second and third layer from the top in Fig. E5, are in the same antichain cluster. These are often used for scientific analysis but they provide different functionality: scientific functions, data handling and plotting respectively. They share many predecessors, such as `numpy`, since they all need to handle large quantities of numerical values efficiently. It is interesting to see that these three packages are not all at the same height so a height based clustering would not put them together. On the other hand `sphinx` (a tool for producing documentation seen on the right of Fig. E5, coloured in bright green) is at the same height as `pandas`, and `matplotlib` (bright red nodes in Fig. E5, next to `sphinx`) but is placed in a different cluster as the packages share so few common predecessors.

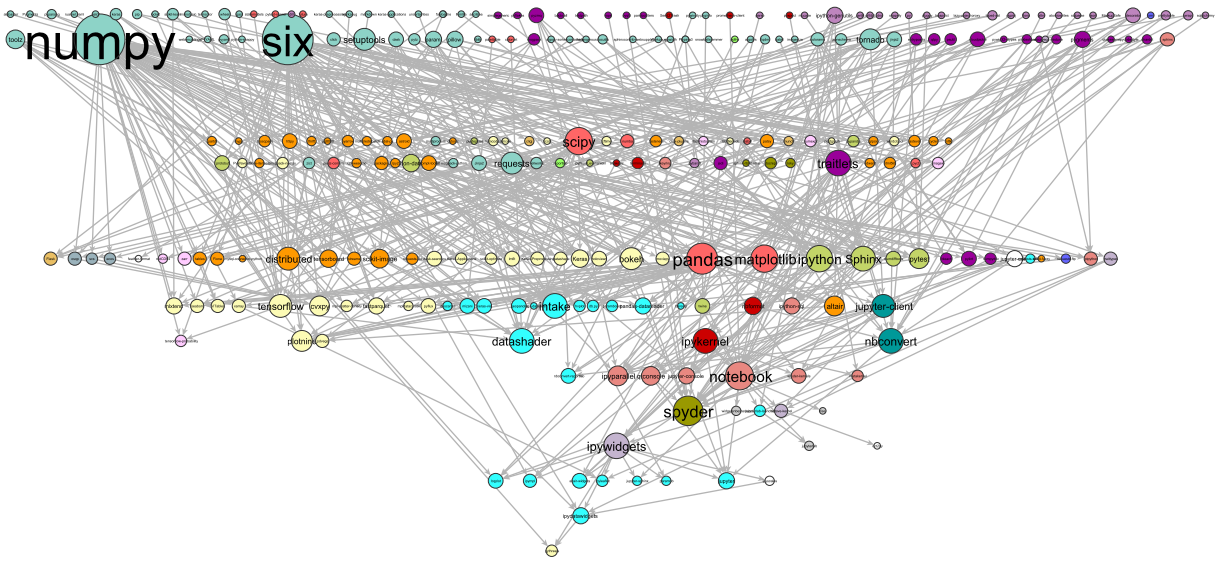


Figure E5: The dependencies of `python` packages on one of the author's machines in 2019. Each node is a package. Links are to the dependent package and are from the parent package needed for the siblings to function, no transitive reduction has been performed. Colours indicate the different antichain communities found using siblinarity based on both successors and predecessors. White indicates a node in a cluster by itself. The size of the node and the size of the labels is related to the degree of each node. The vertical positioning is given by the depth of a node with small variations in the top two depths to improve visibility. An electronic version of this file is available on [7] which will allow readers to see the names of packages with small labels.

F Price Model with Subject Fields

The Price model for citation networks [11] produces a DAG with a fat-tailed (power-law) distribution for the out-degree of nodes in our conventions which represents the citation count of papers. We modify the Price model by assigning each paper to a 'field' and the edges, citations between papers, are biased so they are usually between papers in the same field. While an unrealistic model of citation networks in many ways, it contains three key features of real citation networks: the order of papers imposed by time, the fat-tailed citation count distribution, and the preference of most papers to cite papers within a similar field. We use it as a DAG with a planted partition to enable us to make controlled comparisons between the different community detection approaches discussed.

The model defines a sequence of networks $\mathcal{G}(t)$ where t is a positive integer playing the role of time and which gives us an order to the nodes in our networks. Each graph $\mathcal{G}(t)$ has t nodes with vertex set $\mathcal{V}(t)$ and edge set $\mathcal{E}(t)$. In our notation, the node $u(s)$ is always the node added at step s in the process, so $u(s) \in \mathcal{V}(t)$ provided $0 < s \leq t$.

The nodes in these networks are also partitioned into different fields, that is each node $u(t)$ is in one of F fields. The fields will be labelled by integers between 0 and $(F-1)$ with $f(t) \in \{0, 1 \ldots, (F-1)$ denoting the field of node $u(t)$. This creates a sequence of partitions $\mathfrak{F}(t)$ of our nodes where $\mathfrak{F}(t) = \{\mathcal{F}_f(t) | f \in \{0, 1, \ldots, (F-1)\}\}$ and $\mathcal{F}_f(t) \subseteq \mathcal{V}(t)$. A node $u(s)$, for $0 < s \leq t$, belongs to a element $\mathcal{F}_{f(s)}(t) \in \mathfrak{F}(t)$, the set of papers at time t in the same field $f(s)$ as the paper published at time s.

To create the next graph in the sequence, $\mathcal{G}(t+1)$, we first add a new node $u(t+1)$ to the vertex set, so $\mathcal{V}(t+1) = \mathcal{V}(t) \cup \{u_{t+1}\}$. This new node is assigned to f chosen uniformly at random from the set of F possible field labels.

We now add m directed edges to this new node $v(t+1)$ from existing nodes $u(s)$ where $s < t$. To encode the "cumulative advantage" principle of Price, that is the higher the current citation count of a paper the more likely it is to be cited, we can chose nodes $u(s)$ from the existing nodes $\mathcal{V}(t)$ in the network $\mathcal{G}(t)$ with probability $\Pi^{(\mathrm{CA})}(t, s)$ defined as[3]

$$\Pi^{(\mathrm{CA})}(t, s) \;=\; \frac{k^{\mathrm{out}}(t, s) + 1}{|\mathcal{V}(t)| + |\mathcal{E}(t)|}\,. \tag{33}$$

Here $k^{\mathrm{out}}(t, s)$ is the number of outgoing edges from node $u(s)$ in $\mathcal{G}(t)$, the network at time t. In our conventions, these edges represent citations from later papers to the paper published at time s. The planted partition representing the fields is used on top of the cumulative advantage in $\Pi^{(\mathrm{CA})}(t, s)$ by ensuring that a fraction ϕ of the edges are chosen to lie between nodes in the same field. So the overall probability for choosing an existing node $u(s)$ as the source of an edge to new node $v(t)$ is, to a good approximation[4],

$$\Pi(t, s) \;\approx\; \left(\phi F \delta(f(s), f(t+1)) + (1-\phi)\frac{F}{(F-1)}(1 - \delta(f(s), f(t+1)))\right) \Pi^{(\mathrm{CA})}(t, s)\,. \tag{34}$$

Note that we also impose the constraint that there is at most only one edge between any two nodes and we choose the initial graph to be a transitively complete graph of size $m+1$. Neither of these constraints will have any significant effect on the measurements we make for the large networks we use in our work here.

[3] Other forms linear in $k^{\mathrm{out}}(t, s)$ are also easy to work with (for example see [1]), but these variations are not our focus here. We chose to follow the same form as used in Price's original paper.

[4] For instance, we have assumed that the fraction of nodes in any one field is always $1/F$ and that the degree distribution is the same for all fields at all times. These are good approximations at later times.

G Performance of the algorithm

We studied several metrics of the computational performance of our algorithm. Our current `Python` implementation can be found alongside the data in the Figshare repository [7]. In particular, we looked at the time and memory requirements of the algorithm.

While the runtime depends on the topology of the particular network, we found that our implementation worked for graphs composed of thousands of nodes in a feasible timeframe on a standard desktop computer. In Fig. G6 we show the scaling of time consumption as the input graph size is increased. Here the networks were created using the Price model with subject fields, described in Section 3.2. The longest time to run the code for a network of 9,000 nodes was $1003s$, however, we found the variation in the runtime increases as the input graph becomes larger.

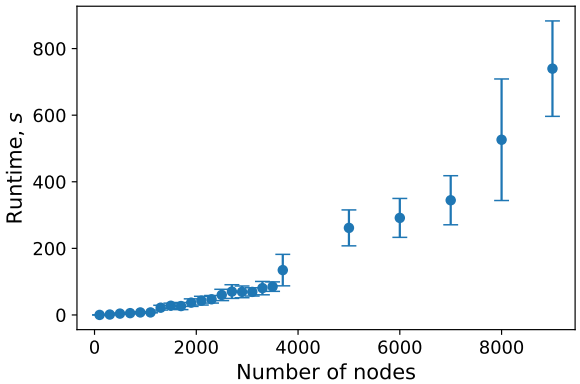


Figure G6: Runtime of an algorithm, for a given size of a network, where the input network is the Price model with fields. In all networks, each node attaches 5 edges to older nodes and 8 out of 10 times a node chooses to connect to another node in its own field. There are three fields in total. For each value of the number of nodes, 10 networks were created.

To estimate the time and memory needed to find siblinarity communities, we recognise there are two aspects to the problem. The first aspect is that we are using a Louvain type algorithm, a type of greedy optimisation, to optimise siblinarity. This appears to run in $O(N\ln(N))$ for a network of N nodes but we have been unable to find a published study to back this up. Given every neighbour of every node is checked, it seems more likely to be $O(E\ln(N))$ for a network of N nodes and E edges. In our case, we use our two-step matrix $\tilde{A}$ which has the same number of nodes but far more edges $\tilde{E}$ than the DAG so the edges will scale as $\tilde{E} \sim \langle k^{\text{out}}\rangle\langle k^{(\text{in})}\rangle N$ in terms of the average in- and out-degree of the DAG of N nodes. In terms of memory, we need to store the information in the similarity matrix $\tilde{A}$ which will scale with the number of edges stored for sparse cases, $O(\tilde{E}) \approx O((\langle k^{\text{out}}\rangle)^2 N))$ (as $\langle k^{\text{out}}\rangle = \langle k^{(\text{in})}\rangle$) with an upper bound of $O(N^2)$ for dense matrices.

The largest difference between the siblinarity optimisation code and the modularity optimisation is that we need to do an extra check for the weak connectivity of two nodes m and n before placing them into the same siblinarity community. In practice, we found that performing this check is feasible by directly checking the `networkx` graph object each time, storing no extra information but require small searches to be performed at each step giving us a time penalty.

In theory we could impose the path constraint needed for antichains in another way and we will use this to provide an estimate for the time requirement of our method. In order to record

the information about node connectivity for a network with adjacency matrix $\mathbf{A}$, we can define a new matrix $\mathbf{P}$ where

$$\mathbf{P} = \sum_{\alpha=1}^{\ell_{\max}} [\mathbf{A}]^{\alpha} \,. \tag{35}$$

Here P_{mn} is the number of paths of any length from node n to mode m. Here $\ell_{\max}$ is the longest path length in the DAG (always finite for a finite DAG and often $O(\ln(N))$). In practice we only need to know if entries are zero or not as only if $P_{mn} = 0$ and $P_{nm} = 0$, can n and m be placed in the same antichain. If we use this approach and define matrix $\mathbf{P}$ then the memory requirements will always be $O(N^2)$ as this is a dense matrix. In terms of time, this extra step to find $\mathbf{P}$ would require $\ell_{\max}$ matrix multiplications if we used matrix multiplication. With upper triangular matrices, this could be fast. However, given that this is a DAG, finding one path (not all paths) from one node to any others, all that is required here, and that will take $O(N + E)$ using a breadth or depth first search in a DAG of N nodes and E edges. So at worst finding the zeros in matrix $\mathbf{P}$, that is finding which pairs of nodes are connected, will take $O((1 + \langle k^{\text{out}} \rangle)N^2)$ in time.

Putting this together, it appears that the memory requirements will always scale as $O(N^2)$ but the time requirements have an lower bound of $O(\langle k^{\text{out}} \rangle N \ln(N))$ and an upper bound of $O(\langle k^{\text{out}} \rangle N^2)$.

H Resolution

As siblinarity optimisation is closely related to modularity, we should consider issues which arise when partitioning a network using modularity optimisation. In particular, to use siblinarity appropriately one must assess which resolutions reveal meaningful siblinarity communities. One way to achieve this in practice is to study the stability of a partition at a given scale (resolution), something studied extensively for algorithms based on modularity optimisation, for example see [12, 13, 14], but this is an issue common to any data clustering method.

To demonstrate how this stability analysis can be performed for our method, we looked at the siblinarity values $S(\mathfrak{A})$ obtained by our implementation of the siblinarity optimisation for a given network across ten different runs for different numbers of communities, here different values of our resolution parameter λ in (9). As Fig. H7 shows, the siblinarity scores tend to decrease with an increase of λ, as expected. However, for a given λ, the scores obtained are very similar (the same colour indicates the same random network). Lastly, we saw a minute variation in the $S(\mathfrak{A})$ values, with standard deviation reaching the maximum of 0.005.

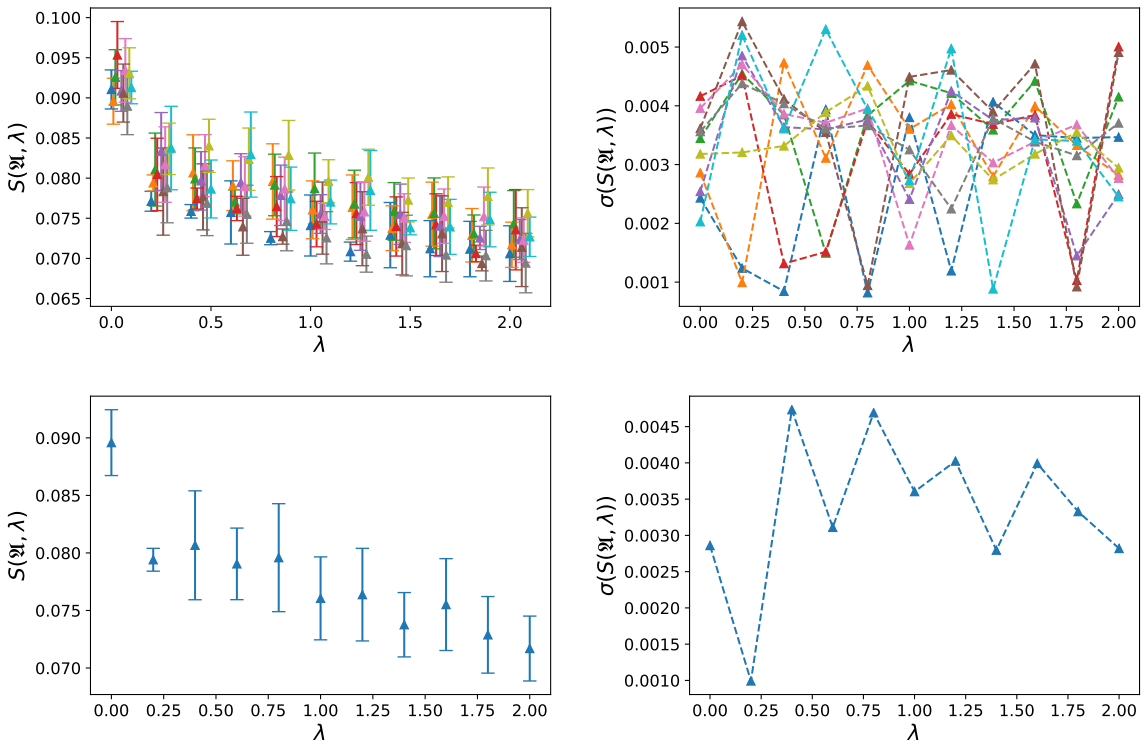


Figure H7: Variations in siblinarity scores $S(\mathfrak{A})$ of (9). On the left, $S(\mathfrak{A})$ for a given network is shown for a variety of λ for multiple networks (top) and a single network (bottom). On the right, the standard deviation of $S(\mathfrak{A})$ obtained by running the algorithm ten times for each λ is shown for multiple networks (top) and a single network (bottom). The results from each network in the top figures are indicated using one colour and at a slightly shifted value of lambda to aid visualisation. The mean and the standard deviation were obtained by running the code 10 times for a given λ. Each network was created the Price model with subject fields, described in Section 3.2 and has $N = 1000$ nodes. Each node attaches 5 edges to older nodes. 80% of the time a node chooses another node from its field (in total, there are 3 fields). For the plot of standard deviation the dashed lines are only shown to guide the eye.

References

[1] Newman, M.: Networks: An Introduction. Oxford University Press, Oxford (2010)

[2] Wikipedia: Family tree of the Greek gods. https://en.wikipedia.org/wiki/Family_tree_of_the_Greek_gods

[3] Satuluri, V., Parthasarathy, S.: Symmetrizations for clustering directed graphs. In: Proceedings of the 14th International Conference on Extending Database Technology - EDBT/ICDT '11. ACM Press, New York, NY, USA (2011)

[4] Newman, M.E.J., Girvan, M.: Finding and evaluating community structure in networks. Phys. Rev. E **69**(2), 026113 (2004)

[5] Reichardt, J., Bornholdt, S.: Statistical mechanics of community detection. Phys. Rev. E **74**(1), 016110 (2006)

[6] Blondel, V.D., Guillaume, J.-L., Lambiotte, R., Lefebvre, E.: Fast unfolding of communities in large networks. Journal of Statistical Mechanics: Theory and Experiment **2008**(10), 10008 (2008)

[7] Vasiliauskaite, V., Evans, T.S.: Data for "Making Communities Show Respect for Order" Paper. doi:10.6084/m9.figshare.9725159. https://figshare.com/s/3ecc2bd6919a64916f44

[8] Python Software Foundation: Python programming language. https://www.python.org/

[9] Python Packaging Authority (PyPA): Python Package Index (PyPI). https://pypi.org/

[10] Naik, V.: pipdeptree. https://github.com/naiquevin/pipdeptree

[11] Price, D.J.d.S.: A general theory of bibliometric and other cumulative advantage processes. J.Amer.Soc.Inform.Sci. **27**, 292–306 (1976)

[12] Fortunato, S.: Community detection in graphs. Physics Reports **486**(3-5), 75–174 (2010). 0906.0612v2

[13] Schaub, M.T., Delvenne, J.-C., Yaliraki, S.N., Barahona, M.: Markov dynamics as a zooming lens for multiscale community detection: non clique-like communities and the field-of-view limit. PLoS ONE **7**(2), 32210 (2012). doi:10.1371/journal.pone.0032210

[14] Lambiotte, R., Delvenne, J.-C., Barahona, M.: Random walks, markov processes and the multiscale modular organization of complex networks. IEEE Transactions on Network Science and Engineering **1**(2), 76–90 (2014). doi:10.1109/tnse.2015.2391998