NetMODE: Network motif detection without Nauty; supplementary material

Xin Li, Douglas S. Stones, Haidong Wang, Hualiang Deng, Xiaoguang Liu, Gang Wang

June 28, 2012

This document is supplementary material for the main paper: “NetMODE: Network motif detection without Nauty.” In particular, we introduce the Reconstruction Conjecture and discuss the generation of random similar graphs. We also describe a basic method for calling NetMODE from R, which allows the user to analyze the data in their own way. We also include the results of some experiments not described in the main text.

Contents

1 Graph theory 2
  1.1 Introduction ......................................................... 2
  1.2 The Reconstruction Conjecture .................................. 4
    1.2.1 Application in NetMODE .................................. 5
  1.3 Similar graphs .................................................... 5
    1.3.1 Kavosh and FanMod ....................................... 6
  1.4 Configuration Model ............................................. 7
    1.4.1 Uniformity ................................................ 9
    1.4.2 Comparison graphs in NetMODE ......................... 10

2 Using NetMODE 11
  2.1 R ................................................................. 11
  2.2 Command line .................................................. 15

3 Further testing of NetMODE 16
  3.1 Number of comparison graphs ................................ 16
  3.2 E. Coli transcription factor network ......................... 16
  3.3 Output comparison with Kavosh and FanMod ............... 18
    3.3.1 Bias in FanMod ....................................... 18
1 Graph theory

In this section, we give an introduction to the graph theory concepts required to understand the concepts in the main text, along with an introduction to the Reconstruction Conjecture and the variant of the Configuration Model used for generating similar graphs uniformly at random.

1.1 Introduction

A graph is a collection of nodes (or vertices) together with a collection of edges joining pairs of nodes. Typically graphs are denoted $G = (V, E)$ where $V = V(G)$ is the set of nodes and $E = E(G)$ is the set of edges. The edges (or arcs) in a directed graph (or digraph) have an order, that is, an edge $(i, j) \in E(G)$ is different to $(j, i) \in E(G)$. Thus it is possible for both $(i, j)$ and $(j, i)$ to occur in the same directed graph, which we refer to as a bidirectional edge. We tend to think of a directed edge $(i, j)$ as coming out of node $i$ and going into node $j$. In an undirected graph, the edges do not have an order, so an edge $\{i, j\} = \{j, i\}$. In either case, $i$ and $j$ are called the endpoints.

An undirected graph can be considered to be a directed graph in which undirected edges $\{i, j\}$ are replaced by directed edges $(i, j)$ and $(j, i)$. This concept is illustrated below. If both edges $(i, j)$ and $(j, i)$ occur in the graph, then we typically think of them combined into a single bidirectional edge. NetMODE focuses on directed networks since they are therefore more general.

Loops (or self-loops) are where an edge $(i, i)$ from a vertex to itself exists. We caution the reader that, for practical reasons, NetMODE ignores loops. A more general class of graphs include parallel edges, where edges $(i, j)$ occur more than once. NetMODE also ignores parallel edges.

In standard usage, a subgraph of $G = (V, E)$ is a graph $H = (V', E')$ for which $V' \subseteq V$ and $E' \subseteq E$ and the edges in $E'$ have endpoints in $V'$. A subgraph is called induced if $E' = \{(i, j) \in E : i \in V' \text{ and } j \in V'\}$, that is, it includes every edge in the original graph whose endpoints are both in $V'$. A graph is called connected if there is a path between any two vertices ignoring edge directions (in graph theory, this is referred to as weakly connected).

Most network motif detection software count only connected induced subgraphs (and many papers omit mention of the requirement that they be connected and induced). MODA \[18\], RAGE \[12, 13\], and NeMo \[10\] are exceptions here, as they can perform non-induced subgraph enumeration (see also \[1\]).

Below are three graphs that highlight the difference. The graph in the center is a subgraph of the left graph, however it is neither connected nor induced. The graph on the
right is a (weakly) connected and induced subgraph.

\[ \begin{array}{ccc}
\bullet & \rightarrow & \bullet \\
\bullet & \rightarrow & \bullet \\
\bullet & \rightarrow & \bullet
\end{array} \]

A directed graph; a subgraph; an induced subgraph;

A graph is called \textit{labeled} if its vertices have been assigned distinct \textit{labels}; often these labels are \( \{0, 1, \ldots, n-1\} \) for an \( n \)-node graph. A permutation \( \alpha \) can be used to “act on” a labeled graph \( G \), specifically, \( \alpha G \) is the graph formed by relabeling each vertex \( i \) in \( G \) by \( \alpha(i) \). We call \( \alpha \) an \textit{isomorphism} and say \( G \) and \( \alpha G \) are \textit{isomorphic}. In fact, if \( G \) has \( |V| = n \) vertices, then there are \( n! \) such isomorphisms, together forming the \textit{symmetric group} \( S_n \) on \( n \) elements. The \textit{isomorphism class} of \( G \) is the set of labeled graphs

\[ \{\alpha G : \alpha \in S_n\} \].

Note that an isomorphism class might not have size \( n! \) since it is possible that \( G = \alpha G \) for some non-trivial \( \alpha \).

There are two common ways to represent an isomorphism class of graphs. Firstly, a graph is called \textit{unlabeled} if its vertices are indistinguishable (except through their adjacencies). In this case, we omit labels when drawing the graph. Assigning vertex labels to an unlabeled graph in all possible ways yields the whole isomorphism class.

An example of an unlabeled graph and its corresponding isomorphism class is given below.

\[ \begin{array}{ccc}
\bullet & \rightarrow & \bullet \\
\bullet & \rightarrow & \bullet \\
\bullet & \rightarrow & \bullet
\end{array} \]

The second way is by choosing a representative element of the isomorphism class. Let \( \Gamma \) be a set of labeled graphs, e.g. the set of \( k \)-node labeled directed graphs. Then a function \( C \) whose domain is \( \Gamma \) is called a \textit{canonical labeling} if \( C(G) = C(G') \) for \( G, G' \in \Gamma \) if and only if \( G \) and \( G' \) are isomorphic. The \textit{canonical label} of a graph \( G \) therefore determines which isomorphism class it belongs to. Simply put, if we input two isomorphic graphs, we get the same output, and if we input two non-isomorphic graphs, we get different outputs.

In NetMODE, we take the canonical label approach. We will now give the details of our canonical labeling. An \textit{adjacency matrix} of a graph \( G = (V, E) \) on \( |V| = n \) nodes is an \( n \times n \) matrix whose \((i, j)\)-th entry is 1 if \((i, j) \in E(G)\) and 0 otherwise. From the adjacency matrix we can obtain a binary number by reading from left-to-right and top-to-bottom. We can convert this to a decimal number, which we call the \textit{graphID}. Therefore every number from 0 to \( 2^{n^2} - 1 \) is the graphID of a labeled digraph on \( n \) vertices (although, some will have loops). We define the canonical label \( C(G) \) to be the minimum graphID amongst all labeled graphs in the isomorphism class containing \( G \). In the example below, the canonical label would be 56044064, which can be found by computing the graphID of every isomorphic graph (there are 720 of them).
Since NetMODE ignores loops, the adjacency matrix of all graphs is assumed to have an all-0 main diagonal, so to save space, NetMODE discards this redundant information. Every graph therefore has an internal graphID, formed in the same way as the normal graphID, but the main diagonal is skipped.

Keep in mind that the number of nodes of the graph is somehow implicit in this type of canonical label. To illustrate, the $k$-node null graph (which has $k$ nodes and no edges) has the canonical label 0, regardless of $k$. For the purpose of network motif detection, this is not a problem since we consistently stick to a fixed $k$.

### 1.2 The Reconstruction Conjecture

In this section we deal exclusively with unlabeled graphs. Given a graph $G = (V,E)$ on $|V| = n$ vertices, we can construct another graph from $G$ by deleting one of the vertices $v \in V$ along with the edges which have $v$ as an endpoint; this graph is denoted $G \setminus v$. The multiset $\{G \setminus v : v \in V\}$ is called the deck of $G$ and any element in the deck of $G$ is called a card. Note that a deck is a multiset, i.e. it can contain multiple copies of the same element.

The Reconstruction Conjecture states that, for undirected graphs, if two graphs on three or more vertices have the same deck, then they are isomorphic [4].

However, for the purposes of NetMODE, we are interested in the variant of the Reconstruction Conjecture for loop-free directed graphs. In fact, it has been known for some time that there are so-called nonreconstructable digraphs [3, 19, 20], that is, the Reconstruction Conjecture is false for digraphs. However, for small $k$ at least, there are very few exceptions to the Reconstruction Conjecture for digraphs. For $k = 6$, we find exactly nine exceptions, which are tabulated below using their canonical labels.
The graph 1162783744, drawn on page 4, has the canonical label 56044064, and by replacing the edge (3, 2) with (2, 3), we obtain a graph with the canonical label 56036128 (i.e. these are the two 6-node graphs in the first exception).

1.2.1 Application in NetMODE

A list of all 5-node loop-free graphs can easily be stored in memory; it can consist of $2^{20}$ ints (32-bit). In the 6-node case, we instead have $2^{30}$ graphs, which is too many to be practically stored in memory. As such, we use the Reconstruction Conjecture for digraphs to essentially change the canonical label for 6-node graphs to their decks of 5-node subgraphs (barring the few exceptions).

We need to quickly determine whether or not a subgraph $H$ is one of the exceptions listed above. This is achieved according to Algorithm 1.

**Algorithm 1** Is $H$ an exceptional subgraph?

1. int $t := H \pmod{7045650}$.
2. if $H = \text{exception_table}[t]$ then
   3. $H$ is an exception.
   4. Increase $\text{exceptionID}[t]$.
5. else
6. $H$ is not an exception (so continue as normal).
7. end if

During the preprocessing phase, we generate and store an array $\text{exception_table}$ of length 7045650. This array is designed so that if $H$ is a connected 6-node subgraph and $\text{exception_table}[t] = H$, where $t = H \pmod{7045650}$, then $H$ is an exceptional subgraph, and we instead increase the value of $\text{exceptionID}[t]$, an array which stores the counts of the exceptional subgraphs. The vast majority of the elements in $\text{exception_table}$ are 0.

1.3 Similar graphs

If $v$ is a vertex in a graph $G = (V, E)$, then its in-degree is the number of edges of the form $(i, v) \in E$. Similarly its out-degree is the number of edges of the form $(v, j) \in E$. 

<table>
<thead>
<tr>
<th>deck</th>
<th>$G$</th>
<th>$G'$</th>
</tr>
</thead>
<tbody>
<tr>
<td>43798</td>
<td>56044064</td>
<td>56036128</td>
</tr>
<tr>
<td>51088</td>
<td>1133217864</td>
<td>1133217802</td>
</tr>
<tr>
<td>549790</td>
<td>1640468230</td>
<td>1640206094</td>
</tr>
<tr>
<td>582550</td>
<td>1657244500</td>
<td>1656986394</td>
</tr>
<tr>
<td>582580</td>
<td>1657252436</td>
<td>1656990300</td>
</tr>
<tr>
<td>1126296</td>
<td>3311111272</td>
<td>3311111210</td>
</tr>
<tr>
<td>1599370</td>
<td>3788211532</td>
<td>3788211470</td>
</tr>
<tr>
<td>1663918</td>
<td>3794604</td>
<td>3856371052</td>
</tr>
<tr>
<td>1666970</td>
<td>3838289726</td>
<td>3856370990</td>
</tr>
</tbody>
</table>
Given an input network $G = (V, E)$, a similar graph is defined to be a graph with the same vertices $V$ and for which each $v \in V$ has the same in-degrees and out-degrees. There are some problems with this definition:

Q1 How can we practically generate similar graphs?

Q2 How are bidirectional edges treated?

1.3.1 Kavosh and FanMod

The answer to the first question (Q1) was addressed in Kavosh and FanMod (among other programs) by a switching method, similar to the method described in [15]. The switching operation is quite simple: two edges $(a, c)$ and $(b, d)$ are chosen, and are replaced by $(a, d)$ and $(b, c)$ provided no loops or multiple edges are introduced.

![Diagram of edge switching](image)

Given a vertex $v \in V$ in a graph $G = (V, E)$, the neighborhood, denoted $N(v)$, of $v$ is the set of vertices $i$ for which either $(i, v) \in E$ or $(v, i) \in E$ (or both). Vertices in $N(v)$ are called the neighbors of $v$. The out-neighborhood, denoted $N_{out}(v)$, is the set of vertices $i$ for which $(v, i) \in E$. Vertices in $N_{out}(v)$ are called the out-neighbors of $v$.

Rather than uniformly choose the two edges in the switch, Kavosh iterates through each vertex $a$ several times. Once $a$ is chosen, Kavosh then picks another vertex $b$ at random ($a \neq b$), and chooses random out-neighbors $c \in N_{out}(a)$ and $d \in N_{out}(b)$. Kavosh will apply the switch provided the following conditions are satisfied:

- $(a, c) \in E$,
- $c \neq b$,
- $(b, d) \in E$,
- $d \neq a$,
- $(b, c) \not\in E$,
- $(c, a) \not\in E$,
- $(a, d) \not\in E$,
- $(d, b) \not\in E$.

Consequently, Kavosh will also leave all bidirectional edges fixed, that is, if both $(u, v)$ and $(v, u)$ are present in the input network, then every subsequent comparison graph will have both $(u, v)$ and $(v, u)$. This can result in some undesirable behavior. For example, regardless of the input network, Kavosh would never declare as a motif e.g. the complete directed graph $\vec{K}_3$, since it will occur no fewer times in the input graph than in any comparison graph.

However, $\vec{K}_3$ (also known as the “fully connected triad”) was considered a motif in a world wide web network in [16]. Furthermore, it is possible to satisfy Kavosh’s conditions while e.g. $(d, a) \in E$, in which case, after the switch is performed, a bidirectional edge between $a$ and $d$ is created, and will remain for each subsequent comparison graph generated.

Importantly, the switches applied by Kavosh are not always reversible. This can result in misleading significance levels. Figure 6 (on page 22) compares the Z-scores returned of Kavosh, FanMod, and NetMODE, running the same experiment. Aside from “no regard” (NR) mode (which is not a suitable switching method in this case), we see that Kavosh’s results are (a) significantly larger than the other results and (b) vary considerably more.
Figure 1: Kavosh’s switching method will, at some point, change the left graph into the right graph, which subsequently will never be modified.

The reader can verify the problem with Kavosh’s switching method by running Kavosh on the graph on the left hand side in Figure 1, where only one possible switch can be performed by Kavosh. Eventually, Kavosh will make this switch, thereby generating a bidirectional edge which will never be modified. This effect would be apparent from Kavosh’s results, however, it is obscured by two bugs: (a) the IDs are mislabeled and (b) to compute the concentrations in the comparison graphs, Kavosh incorrectly divides by the total number of graphs in the input network (when it should be the total number of graphs in the corresponding comparison graph).

Note that all of the bugs we have found in Kavosh could be easily repaired.

FanMod, on the other hand, allows the user to choose one of three different ways of handling bidirectional edges [22].

1. No regard. Bidirectional edges are not preserved.
2. Global constant. The total number of bidirectional edges is fixed.
3. Local constant. The number of adjoining bidirectional edges at each vertex is preserved.

1.4 Configuration Model

We will now introduce the version of the Configuration Model used in NetMODE, and verify its correctness. We refer the reader to [17] for more details on the Configuration Model. We call this algorithm the Mixed Graph Configuration Model. We will consider a mixed graph as a graph that can have either single-directional or undirected edges between nodes (and no parallel edges). There are other definitions of “mixed graph” around. A digraph can be interpreted as a mixed graph by replacing each bidirectional edge with an undirected edge, and by our definition, the converse holds too. While we will not deal with mixed graphs directly, we will treat bidirectional edges as distinct from single-directional edges, thereby motivating the title “Mixed Graph Configuration Model”.

In “local constant” mode, we define a graph as similar to $G$ if it (a) has the same vertices, (b) each vertex has the same in-degree and out-degree as $G$, and (c) each vertex has the same bidirectional degree (i.e. is the endpoint of the same number of bidirectional edges as in $G$).

The Mixed Graph Configuration Model proceeds as follows.
Stage 1. Suppose \( \{v_{2i-1}, v_{2i}\}_{1 \leq i \leq k} \) is a list of bidirectional edges in \( G \). Let \( \alpha \) be a permutation of \( \{1, 2, \ldots, 2k\} \) chosen uniformly at random. Then the candidate edges for the random graph are \( \{v_{\alpha(2i-1)}, v_{\alpha(2i)}\}_{1 \leq i \leq k} \). If these candidates contain a loop or parallel edge, we restart the algorithm.

![Diagram](image1)

Figure 2: Illustrating Stage 1 of the Mixed Graph Configuration Model. The graph on the left hand side lists the bidirectional edges. After randomly permuting the labels, we obtain the middle graph. The graph on the right hand side is generated from the graph in the middle by identifying vertices.

Stage 2. At this stage, we collect the list of single-directional edges, which we label as \( (x_i, y_i)_{1 \leq i \leq k} \). Let \( \beta \) be a permutation of \( \{1, 2, \ldots, k\} \) chosen uniformly at random. Then the candidate edges for the random graph are \( (x_i, y_{\beta(i)})_{1 \leq i \leq k} \). If these candidates edges contain a loop, parallel edge (including the candidates at Stage 1), or bidirectional edge, we restart the algorithm.

![Diagram](image2)

Figure 3: Illustrating Stage 2 of the Mixed Graph Configuration Model. The graph on the left hand side is generated from a random permutation. The graph on the right hand side is generated from the graph on the left by identifying vertices.

Return graph. We return the graph with bidirectional edges \( \{v_{\alpha(2i-1)}, v_{\alpha(2i)}\}_{1 \leq i \leq k} \) and single-directional edges \( (x_i, y_{\beta(i)})_{1 \leq i \leq k} \).
Note that generating random permutations uniformly at random can be efficiently achieved by using a Fisher-Yates Shuffle. Note also that it is important to restart from scratch whenever a clash arises. If, for example, we restarted only Stage 2 when a Stage 2 clash arises, then the resulting distribution would be non-uniform (since the probability of restarting Stage 2 could vary with the results of Stage 1).

1.4.1 Uniformity

We will now show that the Mixed Graph Configuration Model indeed generates similar graphs uniformly at random.

Let \( \text{out}(v) \) and \( \text{in}(v) \) respectively denote the out-degree and in-degree of vertex \( v \) in \( G \) (including only singly directed edges). Let \( \text{deg}(v) \) denote the degree of vertex \( v \) (including only bidirectional edges).

**Theorem 1.** Let \( H \) be a graph similar to the input network \( G \). Then the probability that the Mixed Graph Configuration Model generates \( H \) is

\[
\left( \frac{2^r r! \prod_v \text{deg}(v)!}{(2r)!} \right) \cdot \left( \frac{\prod_v \text{in}(v)! \text{out}(v)!}{s!} \right),
\]

where \( r \) is the number of bidirectional edges in \( G \) and \( s \) is the number of single-directional edges in \( G \). Therefore, the Mixed Graph Configuration Model generates random graphs uniformly at random.

**Proof.** This will be largely a “proof by example” designed to illustrate the concepts behind the proof (aimed at a non-mathematical audience). An actual proof readily follows from the theory of group actions, in particular, the Orbit-Stabilizer Theorem.

Firstly, we observe that, since \( H \) and \( G \) are similar graphs, \( r \) is the number of bidirectional edges in \( H \) and \( s \) is the number of single-directional edges in \( G \).

**Stage 1.** We will begin with an illustrative example. Consider the case when \( G \) has the bidirectional edges: \( \{1, 2\}, \{1, 3\}, \{1, 4\}, \{2, 4\}, \{2, 5\} \). From this set, we construct a list \( \ell = (1, 2, 1, 3, 1, 4, 2, 4, 2, 5) \). We can imagine that the algorithm would randomly permute \( \ell \) (using \( \alpha \)) then return a list of candidate bidirectional edges from that list. For example, if \( (2, 1, 1, 5, 1, 4, 2, 4, 2, 3) \) was obtained by the random permutation, then the candidates would be \( \{2, 1\}, \{1, 5\}, \{1, 4\}, \{2, 4\}, \{2, 3\} \). This is a valid list of candidates since it contains no loops nor parallel edges. However, the order in which the candidates are listed does not matter, nor does the order in which the elements in each candidate is listed (since the edges are bidirectional). So, of all possible distinct lists formed by permuting the elements of \( \ell \), we find exactly \( 2^5 \cdot 5! \) of them would give rise to the above set of valid candidates. In fact, this is true in general; the number of distinct lists that give rise to a valid set of candidates is \( 2^r r! \). (They are all distinct since a valid list of candidates must not have loops or parallel edges.)

There is also a second point to raise. There are permutations \( \alpha \) that do not give distinct lists. For example, if we swap two of the 1’s in \( \ell \), we obtain \( \ell \) back again. In fact, we can get \( \ell \) back again via \( 3! \cdot 3! \cdot 1! \cdot 2! \cdot 1! = 72 \) different permutations. Moreover, of the \( 10! \) lists that are obtained by permuting \( \ell \), we will find exactly 72 copies of every list. In general,
we will find $\prod_v \deg(v)!$ copies of every list (since $\deg(v)$ is the number of copies of $v$ in the list). We invoke the Orbit-Stabilizer Theorem here.

We conclude that there are exactly $2^r r! \prod_v \deg(v)!$ permutations $\alpha$ that give rise to any given valid list of candidate edges. Hence the probability that a given valid list of candidates (and thus the bidirectional component of $H$) is returned after one iteration of Stage 1 is

$$\frac{2^r r! \prod_v \deg(v)!}{(2r)!}.$$

**Stage 2.** Again we will begin with an example. Consider the case when $G$ has the singly directed edges: $(1, 5), (2, 5), (2, 6), (5, 3), (5, 6), (5, 7)$. We construct two lists $x_G = (1, 2, 5, 5, 5)$ and $y_G = (5, 5, 6, 3, 6, 7)$ from the elements in the first and second coordinates, respectively. This time, we randomly permute $y_G$ (using $\beta$) and return the pairs $(x_G[i], y_G[\beta(i)])$ as the candidate edges.

In the same vein as with $G$, the similar graph $H$ comes with a set of singly directed edges, that also gives rise to two lists $x_H$ and $y_H$. In fact, we can assume $x_H = x_G$ (otherwise, we can re-label $H$ so that $x_H = x_G$, and if this is impossible then $H$ and $G$ are not similar). We also know that $y_H$ and $y_G$ have the same elements, and the same number of copies of each element (again, otherwise $H$ and $G$ are not similar). So, it is possible to permute $y_G$ to obtain $y_H$.

The number of ways of permuting $y_H$ (or $y_G$, or, in fact, any list with the same multiset of symbols as $y_H$) to obtain $y_H$ back again is $\prod_v \text{in}(v)!$. Additionally, there are $\prod_v \text{out}(v)!$ ways of permuting $x_G$ (or $x_H$) to obtain $x_G$ back again, or equivalently, there are $\prod_v \text{out}(v)!$ distinct lists formed by permuting $y_H$ that would give rise to the list of candidates that match $H$ (since there are no loops or parallel edges). We also invoke the Orbit-Stabilizer Theorem here. We can therefore conclude that there are exactly $\prod_v \text{in}(v)! \cdot \text{out}(v)!$ permutations $\beta$ in which we would obtain the same set of candidate edges as in $y_H$.

**Conclusion.** The event “Stage 2 agrees with $H$” is independent of the event “Stage 1 agrees with $H$”. Thus the probability that $H$ is returned by a single iteration of the Mixed Graph Configuration Model is

$$\Pr(H \text{ returned}) = \Pr(\text{Stage 1 agrees with } H \text{ and Stage 2 agrees with } H)$$

$$= \Pr(\text{Stage 1 agrees with } H) \times \Pr(\text{Stage 2 agrees with } H)$$

$$= \left(\frac{2^r r! \prod_v \deg(v)!}{(2r)!}\right) \left(\prod_v \text{in}(v)! \cdot \text{out}(v)! s!\right).$$

Note that uniformity is not affected by which stage we actually perform first, provided we restart from scratch whenever a clash arises. In some cases, it is faster to perform Stage 2 first so NetMODE chooses between these two options.

**1.4.2 Comparison graphs in NetMODE**

For the switching methods in NetMODE we retain Kavosh’s method for choosing pairs of edges, while allowing the user to choose one of the three switching modes, similar to that
in FanMod, along with a Kavosh-like switching mode, and the Mixed Graph Configuration Model. The following options are therefore possible:

0. **Fixed.** Bidirectional edges are fixed in place.

1. **No regard.** Bidirectional edges are not preserved.

2. **Global constant.** The total number of bidirectional edges is fixed.

3. **Local constant.** The number of adjoining bidirectional edges at each vertex is preserved.

4. **Uniform local constant.** Local constant mode, uniformly at random via the Mixed Graph Configuration Model.

The user should decide for themselves which mode to use. For example, if the user’s network was derived from a source in which bidirectional edges are impossible (e.g. a family tree), local constant (or global constant) would likely give a better comparison (N.B. “no regard” mode may create bidirectional edges).

In Figure 4 we tabulate the possible edge configurations that can be modified by NetMODE. The left hand side of the table indicates the edges between $a, b$ and $c, d$, while the top of the table indicates the edges between $a, c$ and $b, d$. The corresponding cell shows the resulting subgraph after the switch has been performed and indicates under which settings that switch will be performed. The edges between $a$ and $b$ and between $c$ and $d$ (or lack thereof) do not affect the switching process and are not taken into account. Configurations that contain an edge $(a, d)$ or $(b, c)$ would give rise to a parallel edge, so these switches are forbidden and consequently omitted from Figure 4.

2 Using NetMODE

2.1 R

We have provided a basic script that enables calling of NetMODE from R (using the igraph library). To use it, the following line needs to be edited in NetMODER.txt:

```r
NetMODECommand <- "c:/users/dougy/dropbox/NetMODE/NetMODE.exe"
```

to correspond to the user’s NetMODE executable.

Once this is done, assuming igraph is already installed, NetMODE can be initialized by the following command in R:

```r
source("c:/users/dougy/dropbox/NetMODE/NetMODER.txt")
```

where again the directory will need to be changed. After which, the user should be able to call NetMODE from R, such as via:
Figure 4: The 16 possible edge configurations that can be modified by NetMODE.
> g <- KavoshToIGraph("c:/users/dougy/dropbox/NetMODE/networks/ecoli")
Read 2553 items
> z <- NetMODE(g, 3, 10, 0, 1, 1)
> z
[[1]]
[1] 6 12 14 36 38 46 74 78 98 102 110 238

[[2]]
[1] 196 322 553 167 12 1 536 758 2 3 17 23

[[3]]
[[3]][[1]]
[1] 1379 1480 1556 1589 1589 1512 1573 1583 1583 1537

[[3]][[2]]
[1] 3451 3672 3821 3901 3876 3702 3854 3875 3866 3768

[[3]][[3]]
[1] 218 122 54 18 21 91 29 31 30 72

The remaining lines have been snipped for brevity. R calls NetMODE in “verbose” mode, where all of the intermediate data is returned. The data stored in z is defined below.

• z[[1]][j] contains the GraphID of the j-th graph in g. We can look at this graph with:

> GraphIDToGraph(3, z[[1]][2])
Vertices: 3
Edges: 2
Directed: TRUE
Edges:

[0] 2 -> 0
[1] 1 -> 2
> GraphIDToAdjacencyMatrix(3, z[[1]][2])
[,1] [,2] [,3]
[1,] 0 0 0
[2,] 0 0 1
[3,] 1 0 0
> m <- GraphIDToGraph(3, z[[1]][2])
> plot(m, layout = layout.fruchterman.reingold, vertex.size = 40)
• $z[[2]][j]$ contains the number of copies (up to isomorphism) of the $j$-th graph in $g$.

• $z[[3]][[j]][i]$ contains the number of copies of the $j$-th graph in $g$ in the $i$-th comparison graph.

We can check whether or not a normal approximation is appropriate visually via:

```r
> z <- NetMODE(g,3,10000,0,1,1)
> t <- z[[3]][[2]]
> plot(table(t))
```

```

<table>
<thead>
<tr>
<th>t</th>
<th>table(t)</th>
</tr>
</thead>
<tbody>
<tr>
<td>3344</td>
<td>3344</td>
</tr>
<tr>
<td>3535</td>
<td>3535</td>
</tr>
<tr>
<td>3621</td>
<td>3621</td>
</tr>
<tr>
<td>3674</td>
<td>3674</td>
</tr>
<tr>
<td>3727</td>
<td>3727</td>
</tr>
<tr>
<td>3780</td>
<td>3780</td>
</tr>
<tr>
<td>3833</td>
<td>3833</td>
</tr>
<tr>
<td>3886</td>
<td>3886</td>
</tr>
<tr>
<td>3939</td>
<td>3939</td>
</tr>
</tbody>
</table>
```

In this instance $z[[2]][2] = 322$, which is surprisingly small. We could also perform a normality test, e.g.:
> shapiro.test(t[1:5000])

    Shapiro-Wilk normality test

data:  t[1:5000]
  W = 0.8498, p-value < 2.2e-16

The theory of network motifs has been subject to some criticism; for example, see [2, 6, 7, 8, 9, 11] (see also [14]). NetMODE’s R extension allows the user to perform their own statistical analyses. It returns all of the subgraph counts, so can return a lot of data.

2.2 Command line

NetMODE runs much faster via the command line. A typical instance is called via e.g.:

  NetMODE -k 3 -c 1000 -b 0 -e 3 -t 1 < ./networks/ecoli > output.txt

There parameters are defined by:

- We are performing a \( k \)-node subgraph census, where \( k \) is restricted to the range \( 3 \leq k \leq 6 \).
- There are \( c \) comparison graphs.
- There is \( b \) burnin used.
- The edge switching method used is 3: local constant switching mode (see page 11 for the complete list of methods possible).
- We limit NetMODE to use \( t \) threads simultaneously.
- The input network is ./networks/ecoli. This input network, along with other examples are provided with NetMODE.
- The output is written to output.txt.
- Note that “verbose mode”, invoked with -v 1, is intended to be used for interfacing with R (or some user-built package).

A typical line in output.txt would begin:

gID: 46 freq: 1 ave_rand_freq: 0.31 (sd: 0.569)

The gID is the graphID of the \( k \)-node subgraph found in the input network (in this case 46). The frequency is given by freq (in this case 1). The ave_rand_freq is the average frequency of this subgraph in the ensemble of random similar graphs, and sd is its empirical standard deviation. This line continues:

conc: 0.00039 ave_rand_conc: 0.00012 (sd: 0.0002114)
which gives the concentration-based statistics. The concentration of this subgraph is given by \( \text{conc} \) (in this case 0.00039). The \( \text{ave}_\text{rand}_\text{conc} \) is the average concentration of this subgraph in the ensemble of random similar graphs, and \( \text{sd} \) is its empirical standard deviation. This line continues:

\[
\text{f-ZScore}: 1.21 \quad \text{f-pValue}: 0.26100000 \quad \text{c-ZScore}: 1.28 \quad \text{c-pValue}: 0.04500000
\]

which gives measures of statistical significance: (a) the frequency-based Z-score \( \text{f-ZScore} \), (b) the frequency-based \( p \)-value \( \text{f-pValue} \), (c) the concentration-based Z-score \( \text{c-ZScore} \), and (d) the concentration-based \( p \)-value \( \text{c-pValue} \). Note that these four results might be contradictory, so the user should be careful when declaring which subgraphs are motifs.

The user can also invoke the switch \(-s\) which ensures that a line of data is returned for every \( k \)-node connected subgraph encountered, regardless of whether it is an induced subgraph of the input network. This switch only works when \( k \leq 5 \), due to the structure of NetMODE.

The switch \(-k\) must be set. The remainder are optional. If they are omitted, the defaults are 0 comparison graphs, 0 burnin, edge switching method 3 (local constant mode), and 1 thread.

3 Further testing of NetMODE

3.1 Number of comparison graphs

Figure 5 (on page 21) plots the run-time by NetMODE as the number of comparison graphs varies. In Figure 5 we perform (a) a 5-node subgraph census on the E. Coli dataset, (b) a 6-node subgraph census on the E. Coli dataset, and (c) a 6-node subgraph census on the social network. Figure 5 shows that, after the initial hump due to the overhead associated with preprocessing, NetMODE achieves constant speedup vs. Kavosh as the number of comparison graph increases.

3.2 E. Coli transcription factor network

We ran NetMODE on the E. Coli transcription factor network downloadable from the RegulonDB website [5].

The feedforward loop is clearly overrepresented in the network (settings: \( -k 3 \) \( -c 10000 \) \( -b 0 \) \( -e 3 \)):

\[
\text{gID}: 38 \quad \text{freq}: 1141 \quad \text{ave}_\text{rand}_\text{freq}: 574.48 \quad \text{sd}: 55.715
\]

\[
\text{conc}: 0.00403 \quad \text{ave}_\text{rand}_\text{conc}: 0.00202 \quad \text{sd}: 0.0001966
\]

\[
\text{f-ZScore}: 10.17 \quad \text{f-pValue}: 0.00000000 \quad \text{c-ZScore}: 10.25 \quad \text{c-pValue}: 0.00000000
\]

as is the bifan (settings: \( -k 4 \) \( -c 1000 \) \( -b 0 \) \( -e 3 \)):

\[
\text{gID}: 204 \quad \text{freq}: 43741 \quad \text{ave}_\text{rand}_\text{freq}: 28424.20 \quad \text{sd}: 2982.713
\]

\[
\text{conc}: 0.00167 \quad \text{ave}_\text{rand}_\text{conc}: 0.00107 \quad \text{sd}: 0.0001139
\]

\[
\text{f-ZScore}: 5.14 \quad \text{f-pValue}: 0.00000000 \quad \text{c-ZScore}: 5.23 \quad \text{c-pValue}: 0.00000000
\]
both of which were detected as network motifs by Milo et al. [16]

In the case of 5-node motifs, the most significant results are given below (settings: -c 10000 -b 0 -e 3). We include only cases where both the frequency and concentration p-values are at most 0.0001. We exclude cases with low frequencies (all of the motifs occur more than 1000 times). During the computation, around $2 \times 10^{13}$ subgraphs were encountered, and 84% of these graphs belong to the same isomorphism class.

![Diagram of network motifs](image)

- **gID = 101904**
  - frequency Z-score = 77.56

- **gID = 3864**
  - frequency Z-score = 46.99

- **gID = 10158**
  - frequency Z-score = 39.13

- **gID = 10156**
  - frequency Z-score = 27.60
3.3 Output comparison with Kavosh and FanMod

Figure 6 (on page 22) lists the Z-score returned by Kavosh, FanMod, and NetMODE on the E. Coli network. In each case we look for the 4-node graph.

\[ \text{gID = 9150} \quad \text{frequency Z-score = 19.41} \]
\[ \text{gID = 25500} \quad \text{frequency Z-score = 17.18} \]

We make the following observations:

- The LC and ULC values returned by NetMODE are quite close, suggesting that the switching method in LC mode is reasonably close to uniform. Also, the results of FanMod’s LC mode differ considerably from NetMODE’s ULC.
- NetMODE’s GC mode seems to benefit from the use of burnin.

3.3.1 Bias in FanMod

Figure 6, along with the results in the main text, highlight a built-in bias in FanMod. We will now give some more experimental results that indicate the extend to which this bias applies.

If we repeat the experiment in the main text with 100000 comparison graphs. We instead obtain Z-scores of $-43.424$ and $87.669$ in FanMod LC and $-16.59$ and $19.21$ in NetMODE ULC, for the subgraphs with graphID 78 and 238, respectively (E. Coli network, 3-node subgraph census). So the bias is not mitigated by repeating the experiment nor increasing the number of comparison graphs.

For $k = 6$, E. Coli network:
• The subgraph with graphID 1092133754 (or 2182389326) returned a Z-score of 6084.6 in FanMod, 47.29 in NetMODE LC, and 57.44 in NetMODE ULC.

• The subgraph with graphID 1091318638 (or 5386965102) returned a Z-score of 3512.5 in FanMod, 10.61 in NetMODE LC, and 10.97 in NetMODE ULC.

Both of these subgraphs have a single-directional edge. This also shows that FanMod’s bias is not limited to the $k = 3$ case, and can have a huge effect. If fact, this bias is so large that it seems that the most significant results found by FanMod are simply the result of this bias.

We can also observe similar behaviour for undirected input networks, as indicated in Table 7 (on page 22). The graph used in this experiment is the western states power grid of the USA [21] sourced from http://www-personal.umich.edu/~mejn/netdata/.

References


Figure 5: Speedup of NetMODE vs. Kavosh (K-speedup) as the number of comparison graphs varies.
<table>
<thead>
<tr>
<th>N</th>
<th>Kavosh</th>
<th>FanMod, NR</th>
<th>FanMod, GC</th>
<th>FanMod, LC</th>
</tr>
</thead>
<tbody>
<tr>
<td>100</td>
<td>16.65</td>
<td>421.10</td>
<td>687.94</td>
<td>51612</td>
</tr>
<tr>
<td>1000</td>
<td>614.84</td>
<td>58.68</td>
<td>635.67</td>
<td>28861</td>
</tr>
<tr>
<td>10000</td>
<td>929.29</td>
<td>387.14</td>
<td>270.75</td>
<td>28524</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>N</th>
<th>NetMODE, Ff</th>
<th>NetMODE, Fc</th>
<th>NetMODE, Ffb</th>
<th>NetMODE, Fcb</th>
</tr>
</thead>
<tbody>
<tr>
<td>100</td>
<td>-4.39</td>
<td>-3.75</td>
<td>-5.26</td>
<td>2.90</td>
</tr>
<tr>
<td>1000</td>
<td>-3.78</td>
<td>-3.52</td>
<td>-3.82</td>
<td>2.23</td>
</tr>
<tr>
<td>10000</td>
<td>-3.92</td>
<td>-3.63</td>
<td>-3.55</td>
<td>2.22</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>N</th>
<th>NetMODE, NRf</th>
<th>NetMODE, NRc</th>
<th>NetMODE, NRfb</th>
<th>NetMODE, NRcb</th>
</tr>
</thead>
<tbody>
<tr>
<td>100</td>
<td>4182.20</td>
<td>2967.55</td>
<td>1890.59</td>
<td>19666.20</td>
</tr>
<tr>
<td>1000</td>
<td>7695.49</td>
<td>3656.28</td>
<td>5170.80</td>
<td>35777.54</td>
</tr>
<tr>
<td>10000</td>
<td>4667.33</td>
<td>5068.59</td>
<td>5391.79</td>
<td>21445.34</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>N</th>
<th>NetMODE, GCf</th>
<th>NetMODE, GCc</th>
<th>NetMODE, GCfb</th>
<th>NetMODE, GCCb</th>
</tr>
</thead>
<tbody>
<tr>
<td>100</td>
<td>-0.74</td>
<td>-0.27</td>
<td>-0.16</td>
<td>0.04</td>
</tr>
<tr>
<td>1000</td>
<td>0.57</td>
<td>0.49</td>
<td>0.69</td>
<td>1.09</td>
</tr>
<tr>
<td>10000</td>
<td>0.80</td>
<td>0.70</td>
<td>0.76</td>
<td>1.23</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>N</th>
<th>NetMODE, Lcf</th>
<th>NetMODE, Lcc</th>
<th>NetMODE, LCfb</th>
<th>NetMODE, LCCb</th>
</tr>
</thead>
<tbody>
<tr>
<td>100</td>
<td>-0.98</td>
<td>-1.06</td>
<td>-1.02</td>
<td>0.24</td>
</tr>
<tr>
<td>1000</td>
<td>-0.95</td>
<td>-0.99</td>
<td>-1.05</td>
<td>0.23</td>
</tr>
<tr>
<td>10000</td>
<td>-0.98</td>
<td>-1.00</td>
<td>-0.98</td>
<td>0.14</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>N</th>
<th>NetMODE, ULCf</th>
<th>NetMODE, ULCC</th>
</tr>
</thead>
<tbody>
<tr>
<td>100</td>
<td>-0.80</td>
<td>-1.01</td>
</tr>
<tr>
<td>1000</td>
<td>-1.01</td>
<td>-0.97</td>
</tr>
<tr>
<td>10000</td>
<td>-0.99</td>
<td>-0.98</td>
</tr>
</tbody>
</table>

Figure 6: Z-scores returned by Kavosh, FanMod, and NetMODE under various settings; E. Coli, 4-node subgraph census.

<table>
<thead>
<tr>
<th>graphID</th>
<th>FanMod LC</th>
<th>NetMODE LC</th>
<th>NetMODE ULC</th>
</tr>
</thead>
<tbody>
<tr>
<td>4382</td>
<td>-6.2336</td>
<td>-5.42</td>
<td>-5.34</td>
</tr>
<tr>
<td>4698</td>
<td>-39.906</td>
<td>-38.89</td>
<td>-38.32</td>
</tr>
<tr>
<td>4958</td>
<td>5405.2</td>
<td>230.59</td>
<td>233.21</td>
</tr>
<tr>
<td>13260</td>
<td>94.678</td>
<td>140.69</td>
<td>136.13</td>
</tr>
<tr>
<td>13278</td>
<td>∞</td>
<td>3178.05</td>
<td>3051.31</td>
</tr>
<tr>
<td>31710</td>
<td>∞</td>
<td>∞</td>
<td>∞</td>
</tr>
</tbody>
</table>

Figure 7: Concentration Z-scores returned by FanMod and NetMODE; power network (undirected), 4-node subgraph census.