CONGA User’s Guide

Steve Gregory

v1.67 – November 6, 2012

What the CONGA program does

The program reads an undirected network (with *n* vertices and *m* edges) from a file and computes a *clustering* — a division of the vertices into *c* possibly overlapping *clusters* — for all values of *c* from 1 to at least *n*. This clustering information is stored in a file, to allow you to view the clustering for any value of *c* without recomputing it each time.

The program also implements the CONGO algorithm, which is the same as CONGA but with a small integer specified for the *h* parameter, and the Peacock algorithm, which transforms the network to an expanded one, to be clustered by a separate clustering algorithm.

Network file format

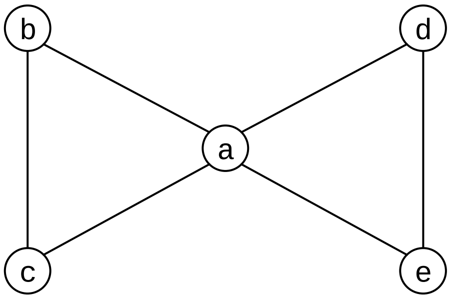
Two alternative formats are accepted for the input network file:

- **CONGA format**. A line containing the name of each vertex (a string), each of which is followed by a line for each of its neighbours, beginning with "--" and a space. Edges *must* be listed in both directions.
- **List of edges format**. A line for each edge: two vertex names (strings) and an optional weight (float), separated by a space or tab. Edges can be listed in only one direction or both.

In both formats, self-edges are ignored. Blank lines and lines beginning with "#" are also ignored.

A vertex name can be any sequence of characters excluding spaces.

For example, the network below left can be represented in two formats as shown:



CONGA format	List of edges
# Network 2	# Network 2
a	a b
-- b	a c
-- c	a d
-- d	a e
-- e	b c
b	d e
-- a	
-- c	
c	
-- a	
-- b	
d	
-- a	
-- e	
e	
-- a	
-- d	

How to run the program

Download the file "conga.jar" and prepare your network in a text file; e.g., "karate.txt". Assuming the network is in CONGA format, you can cluster it by the command:

```
java -cp conga.jar CONGA karate.txt
```

"conga.jar" and the network file may be in any location. If they are not in the current directory, just give the appropriate pathnames instead.

If your network file is in "list of edges" format, add the "-e" option; e.g.:

```
java -cp conga.jar CONGA dolphins-edges.txt –e
```

The output displayed on the screen is in up to four parts:

1. An explanation of the options selected.
2. **Finding clusters**. This shows the steps in the clustering process.
3. **Results**.
4. **Statistics**.

The above commands produce no results, so the "Results" section is empty, and the "Statistics" section give statistics only about the original network and the clustering process.

When a network is clustered, a file is normally produced with a name beginning "clustering-" (e.g., "clustering-karate.txt") that contains the clustering information. If you run one of the above commands for a second time, the clustering will not be performed again, and the "Finding clusters" and "Statistics" sections will be empty. If you want to force the program to recompute clusters (e.g., because the network has changed), use the "-r" option.

One way to get results is using the "-m" option:

```
java -cp conga.jar CONGA karate.txt –m 6
```

This shows, in the "Results" section, some statistics about every clustering containing no more than 6 clusters: namely, Newman's modularity measure. There are several alternative statistics that can be obtained:

-m	Newman's *modularity* measure. This is well-defined for the GN algorithm, in which clusters do not overlap, but is meaningless for the CONGA algorithm.
-mo	The *modularity* measure of Nicosia *et al*., which is extended to handle overlapping clusters.
-vad	*Vertex average degree*: the number of intracluster edges of each vertex, averaged over all vertices.
-ov	*Overlap*: The sum of the sizes of all clusters divided by *n*, the number of vertices in the network.
-dia	*Diameter*: The minimum, average, and maximum cluster diameter.

All of the above options can be used in conjunction with

```
-inc i
```

which means that the statistics will be displayed every *i* clusterings, not for every clustering. This saves time for large networks.

The "-n" option allows you to view the clusters in a particular clustering. For example, to show the division into two clusters:

```
java -cp conga.jar CONGA karate.txt –n 2
```

This shows, in the "Results" section, the size (number of vertices) of each of the clusters. The cluster contents are written to a file with a name beginning "clusters-" (e.g., "clusters-karate.txt"). Each line of the file contains one cluster: a sequence of vertex names separated by spaces.

For very large networks, it is sometimes useful to view only some of the clusters. You can use the "-f" option to see all clusters containing a specified vertex. E.g., to see the cluster(s) containing Donald the dolphin:

```
java -cp conga.jar CONGA dolphins-edges.txt -e –n 2 –f Donald
```

Note: If the network is too large, a runtime error will occur; to solve this, use Java's -Xmx option to increase the maximum heap size, but do not exceed the physical memory available.

Command syntax and options

In general, to run the CONGA program:

```
java –cp conga.jar CONGA filename [options]
```

where options include:

-e

Network file format is a list of edges.
Default: network file format is CONGA native format.

-g *f*

Use file *f* as a filter. This should be a text file with one vertex name on each line. When the network is read in from *filename*, vertices appearing in *f*, and edges to them, are omitted from the network.
Default: all vertices in *filename* are added to the network.

-n *nC*

Find the clustering containing *nC* clusters, or none if *nC* = 0.
Default: *nC* = 0.

-s

Silent operation: don't display the steps in the clustering process.

-cd *t*

If *t* > 0, find the clustering whose mean cluster diameter is the smallest diameter ≥ *t*.
Default: *t* = 0.

`-f` *v*

Show only the cluster(s) containing vertex *v*, in the specified clustering, where *nC* > 0.
Default: show all clusters in the clustering.

`-r`

Recompute the clustering information even if a clustering file exists.

`-mem`

Don't use or create a "`clustering-`" file to store the computed clustering. Keeps the clustering information in memory, which saves disk space and reduces I/O time, but clustering will always run again from the beginning.
Default: keep a "`clustering-`" file.

`-m` *c*

Show the Newman modularity measure for every clustering with at least *nC* and at most *c* clusters.
Default: don't compute or display these statistics.

`-mo` *c*

Show the Nicosia *et al*. modularity measure for every clustering with at least *nC* and at most *c* clusters.
Default: don't compute or display these statistics.

`-vad` *c*

Show the vertex average degree for every clustering with at least *nC* and at most *c* clusters.
Default: don't compute or display these statistics.

`-ov` *c*

Show the overlap for every clustering with at least *nC* and at most *c* clusters.
Default: don't compute or display these statistics.

`-dia` *c*

Show the minimum, mean, and maximum cluster diameter for every clustering with at least *nC* and at most *c* clusters.
Default: don't compute or display these statistics.

`-h` *h*

Run CONGO algorithm, using *h* as the value of the horizon parameter *h*.
Default: *h*=∞ (i.e., same as CONGA).

`-GN`

Run the GN (Girvan and Newman) algorithm. In this algorithm, clusters never overlap.
Default: run the CONGA algorithm.

`-w` *eW*

Weighted mode: accepts weights on edges in network file (in "list of edges" format) and uses these in betweenness computations. *eW* is the weight of edges introduced when splitting vertices: either a positive number or the string "min", "mean", or "max", which respectively denote the minimum, mean, and maximum values of the weights used in the network. The *eW* value is ignored in GN mode.
Default: Weighted mode is off; weights in network file are ignored.

`-fuzzy`

Fuzzy mode: produces a "fuzzy" clustering. Normally, if a vertex belongs to more than one community, it belongs equally to each one. In fuzzy mode, the communities in the clusters file might contain strings of the form "*v*:*b*" where *v* is the vertex name and *b* is *v*'s belonging factor to that community. A vertex name "*v*" alone is equivalent to "*v*:`1`".
In fuzzy mode, in conjunction with the "`-mo`" option, the belonging factors are used in the calculation of the overlap modularity function.
Default: fuzzy mode is off.

`-random` *s*

Random mode. CONGA uses a fast greedy algorithm to find an approximate best split of each vertex. At each step it chooses the best option for that step, but chooses an arbitrary one if there are many best options. Using random mode, CONGA will instead choose a random option if there are more than one. The random number generator uses *s* (a long integer) as seed, so that you can explore different solutions by varying the seed.

`-peacock` *s*

Peacock mode: no clustering. Instead, the network is transformed by splitting vertices only, until the ratio of maximum split betweenness to maximum edge betweenness is ≤ *s*. Outputs the transformed network to files named "`split-`" and creates a vertex names file, named "`vertex-`".
Default: Peacock mode is off.

Using Peacock

The Peacock method comprises three steps:

1. Use the CONGA program in Peacock mode to transform the network.
2. Feed the transformed network into a disjoint clustering program.
3. Postprocess the disjoint clustering to produce an overlapping clustering.

Detailed instructions (for a Unix environment), for four disjoint clustering programs, are given below. The first phase – the Peacock phase – is common to all of them.

The Peacock phase

Given a network in file "*name*`.txt`", run the command

```
java -cp conga.jar CONGA name.txt -peacock 0.1
```

Other CONGA options (e.g., `-h` or `-e`) may be given as well as the `–peacock` option. This command produces three files: "`split-`*name*`.txt`", "`split-`*name*`.csv`", and "`vertex-`*name*`.txt`". The first two contain the transformed network, in different forms, and the last contains the vertex names.

To find out the number of vertices (*n*) in the transformed network, needed for some steps below, look for the line "Final graph size: *n*" in the program's output.

Peacock + CNM algorithm

First calculate *k* = *n*-*c*, where *n* is the transformed network size and *c* is the number of communities desired. Then run the CNM program on the network file, after renaming it as required by the CNM program:

```
cp split-name.txt split-name.pairs
FastCommunityMH -f split-name.pairs -c k
```

where *k* is the value calculated above.

This produces five files, of which only that named "`split-name-fc_a.groups`" is needed. Then postprocess the output by the command (which should be on one line):

```
java  -cp conga.jar CPP split-name-fc_a.groups vertex-name.txt
      clusters-name.txt -cnm
```

This produces the final clustering in "`clusters-name.txt`". It also produces an intermediate form of the clustering in "`$cpp$clusters$.txt`".

Peacock + WT algorithm

First run the WT program on the network in "`split-name.csv`", with output sent to a temporary file *temp*:

```
WT/run.sh --command csv2sdb --input split-name
WT/run.sh --command analyze --input split-name
WT/run.sh --command view-history --input split-name > temp
```

where `WT` is the name of the directory containing the WT programs. This sequence of commands produces six files, which are not needed, plus the *temp* file. Then postprocess the output by the command:

```
java  -cp conga.jar CPP temp vertex-name.txt clusters-name.txt -wt c
```

where *c* is the number of communities desired.

This produces the final clustering in "`clusters-name.txt`". It also produces an intermediate form of the clustering in "`$cpp$clusters$.txt`".

Peacock + PL algorithm

First choose a temporary file *temp* and delete it if it already exists.

Then run the PL program ("`walktrap`") on the network in "`split-name.txt`", with output sent to *temp*:

```
walktrap split-name.txt -o temp -s -d1 -pc
```

where *c* is the number of communities desired.

Then postprocess the output by the command:

```
java  -cp conga.jar CPP temp vertex-name.txt clusters-name.txt -pl
```

This produces the final clustering in "`clusters-name.txt`". It also produces an intermediate form of the clustering in "`$cpp$clusters$.txt`".

Peacock + BGLL algorithm

First run the BGLL program on the network in "`split-name.txt`", with output sent to a temporary file *temp*:

```
BGLL/convert -i split-name.txt -o split-name.bin
BGLL/community split-name.bin -l -1 > split-name.tree
BGLL/hierarchy split-name.tree -l level > temp
```

where BGLL is the name of the directory containing the BGLL programs, and *level* = 0, 1, ... The *level* parameter selects the level in the hierarchy and hence, indirectly, the number of communities in the clustering.

The above sequence of commands produces "split-*name*.bin" and "split-*name*.tree", which are not needed, as well as *temp*. Postprocess the output by the command:

```
java  -cp conga.jar CPP temp vertex-name.txt clusters-name.txt –bgll
```

This produces the final clustering in "clusters-*name*.txt". It also produces an intermediate form of the clustering in "$cpp$clusters$.txt".