NAME

mcl − The Markov Cluster Algorithm, aka the MCL algorithm.

This program implements **mcl**, a cluster algorithm for graphs. A single parameter controls the granularity of the output clustering, namely the **-I** *inflation* option described further below. In standard usage of the program this parameter is the only one that may require changing. By default it is set to 2.0 and this is a good way to start. If you want to explore cluster structure in graphs with MCL, vary this parameter to obtain clusterings at different levels of granularity. A good set of starting values is 1.4, 2, 4, and 6.

The program has a rather large set of options. Except for **-I** none affects the clustering method itself. The other options are for a variety of aspects, such as study of the underlying MCL process (i.e. dumping of iterands), network preprocessing (incorporated for efficiency), resource allocation options (for large-scale analyses), output naming and placement, output formatting, setting of verbosity levels, and so on.

Network construction and reduction techniques should not be considered as part of a clustering algorithm. Nevertheless particular techniques may benefit particular methods or applications. In mcl many transformations are accessible through the **-tf** option. It can be used for edge weight transformations and selection, as well as transformations that act on a graph as a whole. It is for example possible to remove edges with weight below 0.7 by issuing **-tf 'gq(0.7)'**, where the quotes are necessary to prevent the shell from interpreting the parentheses. The option accepts more complicated sequences, such as **-tf 'gq(0.7),add(-0.7)'**. This causes all remaining edge weights to be shifted to the range [0-0.3], assuming that the input contains correlations. Many more transformations are supported, as documented in **mcxio(5)**. Examples of graph-wide transformations are `'#knn(<num>)'` and `'#ceilnb(<num>)'`. The first only keeps those edges that occur in the list of top-`<num>` edges of highest weight in both of its incident nodes. The second removes edges from nodes of highest degree first, proceeding until all node degrees satisfy the given threshold. The **-pi** (pre-inflation) option can be used to increase the contrast in edge weights. This may be useful when clusterings are coarse and fine-grained clusterings are difficult to obtain.

GETTING STARTED

There are two main modes of invocation. The most accessible is *label mode* which assumes a format alternatively called label input or ABC-format. The input is then a file or stream in which each line encodes an edge in terms of two labels (the 'A' and the 'B') and a numerical value (the 'C'), all separated by white space. The most basic example of usage is this:

mcl `<-|fname>` **--abc** **-o** *fname-out*

The output is then a file where each line is a cluster of tab-separated labels. If clustering is part of a larger workflow where it is desirable to analyse and compare multiple clusterings, then it is a good idea to use native mode rather than ABC mode. The reason for this is that native mode is understood by all programs in the mcl suite. It is a more stringent and unambiguous format, and hence more suitable for data exchange. The reader is refered to **clmprotocols(5)** for more information.

SYNOPSIS

The example invocation below assumes matrix input, as explained above and described in the **mcxio(5)** section. Switching to label mode requires the input file to be in ABC-format and the addition of the **--abc** option.

mcl <-|fname> **[-I** <num> (*inflation*)**]** **[-o** <str> (*fname*)**]**

These options are sufficient in 95 percent of the cases or more. The first argument must be the name of a file containing a graph/matrix in the mcl input format, or a hyphen to read from

STDIN. With respect to clustering, the **-I** option is foremost relevant.

The full listing of **mcl** options is shown further below, separated into parts corresponding with functional aspects such as clustering, threading, verbosity, network preprocessing, pruning and resource management, automatic output naming, and dumping.

Baseline clustering options
[-I <num> (*inflation*)**]** **[-o** <fname> (*fname*)**]**

Output options
[-odir <dname> (*directory*)**]** **[--d** (*use input directory for output*)**]**

Input options
[--abc (*expect/write labels*)**]** **[--sif** (*expect/write labels*)**]** **[--etc** (*expect/write labels*)**]** **[--expect-values** (*sif or etc stream contains values*)**]** **[-use-tab** <fname> (*use mapping to write*)**]**

Transform options
[-tf <tf-spec> (*transform input matrix values*)**]** **[-abc-tf** <tf-spec> (*transform input stream values*)**]** **[--abc-neg-log10** (*take log10 of stream values, negate sign*)**]** **[--abc-neg-log** (*take log of stream values, negate sign*)**]** **[-icl** <fname> (*create subgraph on clustering*)**]**

Cache options
[-write-graph <fname> (*write graph*)**]** **[-write-graphx** <fname> (*write transformed graph*)**]** **[-write-expanded** <fname> (*write expanded graph*)**]** **[--write-limit** (*write mcl process limit*)**]**

Input manipulation options
[-pi <num> (*pre-inflation*)**]** **[-ph** <num> (*pre-inflation, max-bound*)**]** **[-if** <num> (*start-inflation*)**]** **[--discard-loops=**<y/n> (*discard y/n loops in input*)**]** **[--sum-loops** (*set loops to sum of other arcs weights*)**]** **[-c** <num> (*reweight loops*)**]**

Clustering processing options
[-sort <str> (*sort mode*)**]** **[-overlap** <str> (*overlap mode*)**]** **[--force-connected=**<y/n> (*analyze components*)**]** **[--check-connected=**<y/n> (*analyze components*)**]** **[--analyze=**<y/n> (*performance criteria*)**]** **[--show-log=**<y/n> (*show log*)**]**

Verbosity options
[-q <spec> (*log levels*)**]** **[-v** <str> (*verbosity type on*)**]** **[-V** <str> (*verbosity type off*)**]** **[--show** (*print (small) matrices to screen*)**]**

Thread options
[-te <int> (*#expansion threads*)**]**

Output file name and annotation options
[-o <str> (*fname*)**]** **[-ap** <str> (*use str as file name prefix*)**]** **[-aa** <str> (*append str to suffix*)**]** **[-az** (*show output file name and exit*)**]** **[-ax** (*show output suffix and exit*)**]** **[-annot** <str> (*dummy annotation option*)**]**

Dump options
[-dump-interval <i:j> (*dump interval*)**]** **[-dump-modulo** <int> (*dump modulo*)**]** **[-dump-stem** <stem> (*dump file stem*)**]** **[-dump** <str> (*type*)**]** **[-digits** <int> (*printing precision*)**]** **[--write-binary** (*write matrices in binary format*)**]**

Info options
[--jury-charter (*explains jury*)**]** **[--version** (*show version*)**]** **[-how-much-ram** k (*RAM upper*

bound)] **[-h** (*most important options*)] **[--help** (*one-line description for all options*)] **[-z** (*show current settings*)] **[-az** (*show output file name and exit*)] **[-ax** (*show output suffix and exit*)] **[--show-schemes** (*show resource schemes*)]

Implementation options
[-sparse <int> (*sparse matrix multiplication threshold*)]

Pruning options
The following options all pertain to the various pruning strategies that can be employed by **mcl**. They are described in the **PRUNING OPTIONS** section, accompanied by a description of the mcl pruning strategy. If your graphs are huge and you have an appetite for tuning, have a look at the following:

[-scheme <int> (*resource scheme*)] **[-resource** <int> (*per-node resource maximum*)] **[-p** <num> (*cutoff*)] **[-P** <int> (*1/cutoff*)] **[-S** <int> (*selection number*)] **[-R** <int> (*recovery number*)] **[-pct** <int> (*recover percentage*)] **[-warn-pct** <int> (*prune warn percentage*)] **[-warn-factor** <int> (*prune warn factor*)]

The first argument of **mcl** must be a file name, but some options are allowed to appear as the first argument instead. These are the options that cause mcl to print out information of some kind, after which it will gracefully exit. The full list of these options is

-z, **-h**, **--help**, **--version**, **--show-settings**, **--show-schemes**, **--jury-charter**.

DESCRIPTION

mcl implements the **MCL algorithm**, short for the **Markov cluster algorithm**, a cluster algorithm for graphs developed by Stijn van Dongen at the Centre for Mathematics and Computer Science in Amsterdam, the Netherlands. The algorithm simulates flow using two simple algebraic operations on matrices. The inception of this flow process and the theory behind it are described elsewhere (see **REFERENCES**). Frequently asked questions are answered in the **mclfaq(7)** section. The program described here is a fast threaded implementation written by the algorithm's creator with contributions by several others. Anton Enright co-implemented threading; see the **HISTORY/CREDITS** section for a complete account. See the **APPLICABILITY** section for a description of the type of graph mcl likes best, and for a qualitative assessment of its speed. **mcl** is accompanied by several other utilities for analyzing clusterings and performing matrix and graph operations; see the **SEE ALSO** section.

The first argument is the input file name, or a single hyphen to read from stdin. The rationale for making the name of the input file a fixed parameter is that you typically do several runs with different parameters. In command line mode it is pleasant if you do not have to skip over an immutable parameter all the time.

The **-I** *f* option is the main control, affecting cluster granularity. In finding good **mcl** parameter settings for a particular domain, or in finding cluster structure at different levels of granularity, one typically runs mcl multiple times for varying values of f (refer to the **-I** *inflation* option for further information).

NOTE MCL interprets the matrix entries or graph edge weights as **similarities**, and it likes **undirected input graphs** best. It can handle directed graphs, but any node pair (i,j) for which w(i,j) is much smaller than w(j,i) or vice versa will presumably have a slightly negative effect on the clusterings output by mcl. Many such node pairs will have a distinctly negative effect, so try to make your input graphs undirected. How your edge weights are computed may affect mcl's performance. In protein clustering, one way to go is to choose the negated logarithm of the BLAST probabilities (see **REFERENCES**).

mcl's default parameters should make it quite fast under almost all circumstances. Taking default parameters, mcl has been used to generate good protein clusters on 133k proteins, taking 10 minutes running time on a Compaq ES40 system with four alpha EV6.7 processors. It has been applied (with good results) to graphs with two million nodes, and if you have the memory (and preferably CPUs as well) nothing should stop you from going further.

For large graphs, there are several groups of parameters available for tuning the mcl computing process, should it be necessary. The easiest thing to do is just vary the **-scheme** option. This triggers different settings for the group of pruning parameters **-p/-P**, **-R**, **-S**, and **-pct**. The default setting corresponds with **-scheme 6**. When doing multiple mcl runs for the same graphs with different **-I** settings (for obtaining clusterings at different levels of granularity), it can be useful to factor out the first bit of computation that is common to all runs, by using the **-write-expanded** option one time and then using **-if** *inflation* for each run in the set. Whether mcl considers a graph large depends mainly on the graph connectivity; a highly connected graph on 50,000 nodes is large to mcl (so that you might want to tune resources) whereas a sparsely connected graph on 500,000 nodes may be business as usual.

mcl is a memory munger. Its precise appetite depends on the resource settings. You can get a rough (and usually much too pessimistic) upper bound for the amount of RAM that is needed by using the **-how-much-ram** option. The corresponding entry in this manual page contains the simple formula via which the upper bound is computed.

Other options of interest are the option to specify threads **-te**, and the verbosity-related options **-v** and **-V**. The actual settings are shown with **-z**, and for graphs with at most 12 nodes or so you can view the MCL matrix iterands on screen by supplying **--show** (this may give some more feeling).

MCL iterands allow a generic interpretation as clusterings as well. The clusterings associated with early iterands may contain a fair amount of overlap. Refer to the **-dump** option, the **mclfaq(7)** manual, and the **clm imac** utility (Interpret Matrices As Clusterings). Use **clm imac** only if you have a special reason; the normal usage of **mcl** is to do multiple runs for varying **-I** parameters and use the clusterings output by mcl itself.

Under very rare circumstances, **mcl** might get stuck in a seemingly infinite loop. If the number of iterations exceeds a hundred and the *chaos* indicator remains nearly constant (presumably around value 0.37), you can force mcl to stop by sending it the ALRM signal (usually done by **kill -s ALRM** *pid*). It will finish the current iteration, and interpret the last iterand as a clustering. Alternatively, you can wait and mcl might converge by itself or it will certainly stop after 10,000 iterations. The most probable explanation for such an infinite loop is that the input graph contains the flip-flop graph of node size three as a subgraph.

The creator of this page feels that manual pages are a valuable resource, that online html documentation is also a good thing to have, and that info pages are way *way* ahead of their time. The **NOTES** section explains how this page was created.

In the **OPTIONS** section options are listed in order of importance, with related options grouped together.

OPTIONS

-I <num> (*inflation*)

Sets the main inflation value to *<num>*. This value is the main handle for affecting cluster granularity. It is usually chosen somewhere in the range [1.2-5.0]. **-I 5.0** will tend to result in fine-grained clusterings, and **-I 1.2** will tend to result in very coarse grained clusterings. Your mileage will vary depending on the characteristics of your data. That is why it is a good idea to test the quality and coherency of your clusterings using **clm dist** and **clm info**.

This will most likely reveal that certain values of **-I** are simply not right for your data. The **clm dist** section contains a discussion of how to use the cluster validation tools shipped with **mcl** (see the **SEE ALSO** section).

With low values for **-I**, like **-I 1.2**, you should be prepared to use more resources in order to maintain quality of clusterings, i.e. increase the argument to the **-scheme** option.

-o <fname> (*output file name*)
-odir <dname> (*output directory name*)
--d (*use input directory for output*)

The default mode of output creation for **mcl** is to create a file name that uses the input file name stripped of any leading path components, augmented with a prefix '`out.`' and a suffix encoding pivotal **mcl** parameters. This will usually be the inflation value which is the argument to the **-I** option. By default the output file is written in the current directory. For example, if the input is named `data/small.mci` for example and inflation is set to three, the output file will be named `out.small.mci.I30`.

This behaviour can be overridden in various ways. The **-o** option simply specifies the output file name, which may include path components that should exist. It is possible to send the clustering to STDOUT by supplying **-o -**. With the **-odir** *<dname>* option **mcl** constructs the output file name as before, but writes the file in the directory *<dname>*. Finally, the option **--d** is similar but more specific in that **mcl** will write the output in the directory specified by the path component of the input file, that is, the directory in which the input file resides.

If either one of **--abc**, **--sif**, **--etc** or **-use-tab** *tab-file* is used the output will be in label format. Otherwise the clustering is output in the mcl matrix format; see the **mcxio(5)** section for more information on this. Refer also to the group of options discussed at **--abc**.

Look at the **-ap** *prefix* option and its siblings for the automatic naming constructions employed by **mcl** if the **-o** option is not used.

-c <num> (*reweight loops*)
--sum-loops (*set loops to sum of other arcs weights*)

With the **-c** *<num>* option, as the final step of loop computation (i.e. after initialization and shadowing) all loop weights are multiplied by **<num>**, if supplied.

--discard-loops=<y/n> (*discard loops in input*)

By default **mcl** will remove any loops that are present in the input. Use **--discard-loops=n** to turn this off. Bear in mind that loops will still be modified in all cases where the loop weight is not maximal among the list of edge weights for a given node.

--abc (*expect/write labels*)
--sif (*expect/write labels*)
--etc (*expect/write labels*)
--expect-values (*expect label:weight format*)
-use-tab <fname> (*use mapping to write*)

These items all relate to label input and/or label output. **--abc** tells **mcl** to expect label input and output clusters in terms of those labels. This simple format expects two or three fields separated by white space on each line. The first and second fields are interpreted as labels specifying source and destination node respectively. The third field, if present, specifies the weight of the arc connecting the two nodes.

The option **--sif** tells **mcl** to expect SIF (Simple Interaction File) format. This format is line based. The first two fields specify the source node (as a label) and the relationship type. An arbitrary number of fields may follow, each containing a label identifying a destination node. The second field is simply ignored by **mcl**. As an extension to the SIF format weights may optionally follow the labels, separated from them with a colon character. It is in this case necessary to use the **--expect-values** option. The **--etc** option expects a format identical in all respects except that the relationship type is not present, so that all fields after the first are interpreted as destination labels.

-use-tab is only useful when matrix input is used. It will use the tab file to convert the output to labels; it does not fail on indices missing from the tab file, but will bind these to generated dummy labels.

-tf <tf-spec> (*transform input matrix values*)
-abc-tf <tf-spec> (*transform input stream values*)
--abc-neg-log10 (*take log10 of stream values, negate sign*)
--abc-neg-log (*take log of stream values, negate sign*)

-tf transforms the values of the input matrix according to **<tf-spec>**. **-abc-tf** transforms the stream values (when **--abc** is used) according to **<tf-spec>**. **--abc-neg-log** and **--abc-neg-log10** imply that the stream input values are replaced by the negation of their log or log10 values, respectively. The reason for their existence is documented in **mcxio(5)**. For a description of the transform language excpected/accepted in **<tf-spec>** refer to the same.

-icl <fname> (*create subgraph on clustering*)

With this option **mcl** will subcluster the provided clustering. It does so by removing, first of all, all edges from the input graph that connect different clusters. The resulting graph consists of different components, at least as many as there are clusters in the input clustering. This graph is then subjected to transformations, if any are specified, and then clustered. The output name is constructed by appending the normal mcl-created file name suffix to the name of the input clustering.

-write-graph <fname> (*write graph*)
-write-graphx <fname> (*write transformed graph*)
-write-expanded <fname> (*write expanded graph*)
--write-limit (*write mcl process limit*)

The first two options are somewhat outdated, in that the prefered way of loading networks is by using **mcxload(1)**. The option **-write-expanded** can be useful for exploring more complicated input transformations that incorporate an expansion step, but is not really relevant for production use. The last option is mainly educational and for analyzing the **mcl** process itself.

-scheme <num> (*use a preset resource scheme*)
-resource <num> (*allow n neighbours throughout*)

There are currently seven different resource schemes, indexed 1..7. High schemes result in more expensive computations that may possibly be more accurate. The default scheme is 4. When **mcl** is done, it will give a grade (the so called *jury synopsis*) to the appropriateness of the scheme used. *A low grade does not necessarily imply that the resulting clustering is bad* - but anyway, a low grade should be reason to try for a higher scheme.

Use the **-resource** *<num>* option to cap for each nodes the number of neighbours tracked during computation at *<num>* nodes.

The **PRUNING OPTIONS** section contains an elaborate description of the way **mcl** manages resources, should you be interested. In case you are worried about the validation of the resulting clusterings, the **mclfaq(7)** section has several entries discussing this issue. The bottom line is that you have to compare the clusterings resulting from different schemes (and otherwise identical parameters) using utilities such as **clm dist**, **clm info** on the one hand, and your own sound judgment on the other hand.

If your input graph is extremely dense, with an average node degree (i.e. the number of neighbours per node) that is somewhere above 500, you may need to filter the input graph by removing edges, for example by using one of **-tf '#ceilnb()'** or **-tf '#knn()'**.

--show-schemes (*show preset resource schemes*)
Shows the explicit settings to which the different preset schemes correspond.

The characteristics are written in the same format (more or less) as the output triggered by **-v pruning**.

-V <str> (*verbosity type off*)
See the **-v** option below.

-v <str> (*verbosity type on*)
These are the different verbosity modes:

pruning
explain
cls
all

-q <spec> (*log levels*)

To make mcl as quiet as can be, add **-q x -V all** to the command line.

The **-q** option governs a general logging mechanism. The format accepted is described in the **tingea.log(7)** manual page.

The other options govern verbosity levels specific to mcl. **-v all** turns them all on, **-V all** turns them all off. **-v** *str* and **-V** *str* turn on/off the single mode *str* (for *str* equal to one of **pruning**, **cls**, or **explain**). Each verbosity mode is given its own entry below.

-v explain
This mode causes the output of explanatory headers illuminating the output generated with the **pruning** verbosity mode.

-v pruning
This mode causes output of resource-related quantities. It has a separate entry in the PRUNING OPTIONS section.

-v cls
This mode (on by default) prints a terse list of characteristics of the clusterings associated with intermediate iterands. The characteristics are **E/V**, **cls**, **olap**, and **dd**. They respectively stand for the number of outgoing arcs per node (as an average), the number of clusters in the overlapping clustering associated with the iterand, the number of nodes in overlap, and the *dag depth* associated with the DAG (directed acyclic graph) associated with the iterand. For more information on this DAG refer to the **-dump** option description in this manual and also **mclfaq(7)**.

Standard log information

m-ie	This gives the ratio of (1) the number of edges after initial expansion, before pruning, to (2) the number of edges of the current iterand.
m-ex	This gives the ratio of (1) the number of edges after expansion (including pruning), to (2) the number of edges of the current iterand.
i-ex	This gives the ratio of (1) the number of edges after expansion (including pruning), to (2) the number of edges of the original input graph.
fmv	This gives the percentage of nodes (matrix columns) for which full matrix/vector computation was used (as opposed to using a sparse technique).

-aa <str> (*append <str> to suffix*)
See the **-ap** option below.

-ap <str> (*use <str> as file name prefix*)
If the **-o** *fname* option is not used, **mcl** will create a file name (for writing output to) that should uniquely characterize the important parameters used in the current invocation of mcl. The default format is **out.fname.suf**, where **out** is simply the literal string `out`, **fname** is the first argument containing the name of the file (with the graph) to be clustered, and where **suf** is the suffix encoding a set of parameters (described further below).

The **-ap** *str* option specifies a prefix to use rather than **out.fname** as sketched above. However, **mcl** will interpret the character '=', if present in *str*, as a placeholder for the input file name.

If the **-aa** *str* option is used, **mcl** will append **str** to the suffix **suf** created by itself. You can use this if you need to encode some extra information in the file name suffix.

The suffix is constructed as follows. The **-I** *f* and **-scheme** parameter are always encoded. Other options, such as **-pi** *f* and **-knn** are only encoded if they are used. Any real argument *f* is encoded using *exactly one* trailing digit behind the decimal separator (which itself is not written). The setting **-I 3.14** is thus encoded as I31. The **-scheme** option is encoded using the letter 's', all other options mentioned here are encoded as themselves (stripped of the hyphen). For example

```
mcl small.mci -I 3 -c 2.5 -pi 0.8 -scheme 5
```

results in the file name `out.small.mci.I30s5c25pi08`. If you want to know beforehand what file name will be produced, use the **-az** option.

-az (*show output file name and exit*)
-ax (*show output suffix and exit*)
If **mcl** automatically constructs a file name, it can be helpful to known beforehand what that file name will be. Use **-az** and mcl will write the file name to STDOUT and exit. This can be used if mcl is integrated into other software for which the automatic creation of unique file names is convenient.

By default mcl incorporates the input file name into the output file name and appends a short suffix describing the most important option settings. Use **-ax** to find out what that suffix is. This can be useful in wrapper pipeline scripts such as clxcoarse.

-annot <str> (*dummy annotation option*)
mcl writes the command line with which it was invoked to the output clustering file. Use this option to include any additional information. MCL does nothing with this option

except copying it as just described.

-te <int> (*#expansion threads*)

Threading is useful if you have a multi-processor system. **mcl** will spawn *k* threads of computation. If these are computed in parallel (this depends on the number of CPUs available to the mcl process) it will speed up the process accordingly.

When threading, it is best not to turn on pruning verbosity mode if you are letting mcl run unattended, unless you want to scrutinize its output later. This is because it makes **mcl** run somewhat slower, although the difference is not dramatic.

-pi <num> (*pre-inflation*)
-ph <num> (*pre-inflation, max-bound*)

If used, **mcl** will apply inflation one time to the input graph before entering the main process. This can be useful for making the edge weights in a graph either more homogeneous (which may result in less granular clusterings) or more heterogeneous (which may result in more granular clusterings). Homogeneity is achieved for values *<num>* less than one, heterogeneity for values larger than one. Values to try are normally in the range `[2.0,10.0]`.

The **-ph** option is special in that it does not rescale columns to be stochastic. Instead, it rescales columns so that the maximum value found in the column stays the same after inflation was applied. There is little significance to this, and what little there is is undocumented.

-if <num> (*start-inflation*)

If used, **mcl** will apply inflation one time to the input graph before entering the main process. The difference with **-pi** is that with the latter option mcl may apply certain transformations after reading in the matrix such as adding or modifying loops. The purpose of the **-if** (mnemonic for *inflation-first*) option is to use it on graphs saved with the **--write-expanded** option and convey to mcl that it should not apply those transformations.

-dump-interval <i:j> (*dump interval*)
-dump-interval *all*

Dump during iterations i..j-1. Use *all* to dump in all iterations. See the **-dump** *str* option below.

-dump-modulo <int> (*dump i+0..i+<int>..*)

Sampling rate: select only these iterations in the dump interval. See the **-dump** *str* option below.

-dump-stem <stem> (*file stem*)

Set the the stem for file names of dumped objects (default *mcl*). See the **-dump** *str* option below.

-dump <str> (*type*)

str is checked for substring occurrences of the following entries. Repeated use of **-dump** is also allowed.

ite
dag
cls
chr
lines

cat

lines and **cat** change the mode of dumping. The first changes the dump format to a line based pairwise format rather than the default mcl matrix format. The second causes all dumped items to be dumped to the default stream used for the output clustering, which is appended at the end.

The **ite** option writes **mcl** iterands to file. The **cls** option writes clusterings associated with mcl iterands to file. These clusters are obtained from a particular directed acyclic graph (abbreviated as DAG) associated with each iterand. The **dag** option writes that DAG to file. The DAG can optionally be further pruned and then again be interpreted as a clustering using **clm imac**, and **clm imac** can also work with the matrices written using the **ite** option. It should be noted that clusterings associated with intermediate iterands may contain overlap, which is interesting in many applications. For more information refer to **mclfaq(7)** and the **REFERENCES** section below.

The **result** option dumps the usual MCL clustering.

The **chr** option says, for each iterand I, to output a matrix C with characteristics of I. C has the same number of columns as I. For each column k in C, row entry 0 is the diagonal or 'loop' value of column k in I *after* expansion and pruning, and *before* inflation and rescaling. Entry 1 is the loop value *after* inflation and rescaling. Entry 2 is the center of column k (the sum of its entries squared) computed *after* expansion and *before* pruning, entry 3 is the maximum value found in that column at the same time. Entry 4 is the amount of mass kept for that column *after pruning*.

The **-ds** option sets the stem for file names of dumped objects (default *mcl*). The **-di** and **-dm** options allow a selection of iterands to be made.

-digits <int> (*printing precision*)
This has two completely different uses. It sets the number of decimals used for pretty-printing **mcl** iterands when using the **--show** option (see below), and it sets the number of decimals used for writing the expanded matrix when using the **-write-expanded** option.

--show (*print matrices to screen*)
Print matrices to screen. The number of significant digits to be printed can be tuned with **-digits** *n*. An 80-column screen allows graphs (matrices) of size up to 12(x12) to be printed with three digits precision (behind the comma), and of size up to 14(x14) with two digits. This can give you an idea of how **mcl** operates, and what the effect of pruning is. Use e.g. **-S 6** for such a small graph and view the MCL matrix iterands with **--show**.

--write-binary (*output format*)
Write matrix dump output in binary mcl format rather than interchange mcl format (the default). Note that **mcxconvert** can be used to convert each one into the other. See **mcxio(5)** and **mcx(1)** for more information.

-sort <str> (*sort mode*)
str can be one of **lex**, **size**, **revsize**, or **none**. The default is 'revsize', in which the largest clusters come first. If the mode is 'size', smallest clusters come first, if the mode is 'lex', clusters are ordered lexicographically, and if the mode is 'none', the order is the same as produced by the procedure used by mcl to map matrices onto clusterings.

-overlap <str> (*overlap mode*)
Mode *keep* causes mcl to retain overlap should this improbable event occur. In theory, **mcl**

may generate a clustering that contains overlap, although this almost never happens in practice, as it requires some particular type of symmetry to be present in the input graph (not just any symmetry will do). Mathematically speaking, this is a conjecture and not a theorem, but the present author wil eat his shoe if it fails to be true (for marzipan values of shoe). It is easy though to construct an input graph for which certain mcl settings result in overlap - for example a line graph on an odd number of nodes. The default is to excise overlapping parts and introduce them as clusters in their own right. It is possible to allocate nodes in overlap to the first cluster in which they occur (i.e. rather arbitrarily), corresponding with mode *cut*.

In mode *split* mcl will put all nodes in overlap into separate clusters. These clusters are chosen such that two nodes are put in the same new cluster if and only if they always occur paired in the clusters of the overlapping clustering.

This option has no effect on the clusterings that are output when using **-dump** *cls* - the default for those is that overlap is not touched, and this default can not yet be overridden.

--force-connected=<y/n> (*analyze components*)
--check-connected=<y/n> (*analyze components*)

If the input graph has strong bipartite characteristics, mcl may yield clusters that do not correspond to connected components in the input graph. Turn one of these modes on to analyze the resultant clustering.

If loose clusters are found they will be split into subclusters corresponding to connected components. With **--force-connected**=*y* mcl will write the corrected clustering to the normal output file, and the old clustering to the same file with suffix `orig`. With **--check-connected**=*y* mcl will write the loose clustering to the normal output file, and the corrected clustering to the same file with suffix `coco`.

These options are not on by default, as the analysis is currently (overly) time-consuming and mcl's behaviour actually makes some sense (when taking bipartite characteristics into account).

--analyze=<y/n> (*performance criteria*)

With this mode turned on, **mcl** will reread the input matrix and compute a few performance criteria and attach them to the output file. Off by default.

--show-log=<y/n> (*show log*)

Shows the log with process characteristics on STDOUT. By default, this mode is off.

--jury-charter (*explains jury*)

Explains how the jury synopsis is computed from the jury marks.

--version (*show version*)

Show version.

-how-much-ram <int> (*RAM upper bound*)

<int> is interpreted as the number of nodes of an input graph. mcl will print the maximum amount of RAM it needs for its computations. The formula for this number in bytes is:

```
2 * c * k * <int>

  2  :  two matrices are concurrently held in memory.
  c  :  mcl cell size (as shown by -z).
<int>:  graph cardinality (number of nodes).
  k  :  MAX(s, r).
  s  :  select number (-S, -scheme options).
  r  :  recover number (-R, -scheme options).
```

This estimate will usually be too pessimistic. It does assume though that the average node degree of the input graph does not exceed k. The **-how-much-ram** option takes other command-line arguments into account (such as **-S** and **-R**), and it expresses the amount of RAM in megabyte units.

-h (*show help*)
Shows a selection of the most important **mcl** options.

--help (*show help*)
Gives a one-line description for all options.

-z (*show settings*)
Show current settings for tunable parameters. **--show-settings** is a synonym.

PRUNING OPTIONS

-p <num> (*cutoff*)
-P <int> (*1/cutoff*)
-S <int> (*selection number*)
-R <int> (*recover number*)
-pct <pct> (*recover percentage*)
After computing a new (column stochastic) matrix vector during expansion (which is matrix multiplication c.q. squaring), the vector is successively exposed to different pruning strategies. The intent of pruning is that many small entries are removed while retaining much of the stochastic mass of the original vector. After pruning, vectors are rescaled to be stochastic again. MCL iterands are theoretically known to be sparse in a weighted sense, and this manoever effectively perturbs the MCL process a little in order to obtain matrices that are genuinely sparse, thus keeping the computation tractable. An example of monitoring pruning can be found in the discussion of **-v pruning** at the end of this section.

mcl proceeds as follows. First, entries that are smaller than *cutoff* are removed, resulting in a vector with at most *1/cutoff* entries. The cutoff can be supplied either by **-p**, or as the inverse value by **-P**. The latter is more intuitive, if your intuition is like mine (P stands for precision or pruning). The cutoff just described is rigid; it is the same for all vectors. The **--adapt** option causes the computation of a cutoff that depends on a vector's homogeneity properties, and this option may or may not speed up mcl.

Second, if the remaining stochastic mass (i.e. the sum of all remaining entries) is less than *<pct>*/100 and the number of remaining entries is less than *<r>* (as specified by the **-R** flag), **mcl** will try to regain ground by recovering the largest discarded entries. The total number of entries is not allowed to grow larger than *<r>*. If recovery was not necessary, mcl tries to prune the vector further down to at most *s* entries (if applicable), as specified by the **-S** flag. If this results in a vector that satisfies the recovery condition then recovery is attempted, exactly as described above. The latter will not occur of course if *<r>* <= *<s>*.

The default setting is something like **-P 4000 -S 500 -R 600**. Check the **-z** flag to be sure. There is a set of precomposed settings, which can be triggered with the **-scheme** *k* option.

k=4 is the default scheme; higher values for *k* result in costlier and more accurate computations (vice versa for lower, cheaper, and less accurate). The schemes are listed using the **--show-schemes** option. It is advisable to use the **-scheme** option only in interactive mode, and to use the explicit expressions when doing batch processing. The reason is that there is *no guarantee whatsoever* that the schemes will not change between different releases. This is because the scheme options should reflect good general purpose settings, and it may become appararent that other schemes are better.

Note that 'less accurate' or 'more accurate' computations may still generate the same output clusterings. Use **clm dist** to compare output clusterings for different resource parameters. Refer to **clm dist** for a discussion of this issue.

-warn-pct <int> (*prune warn percentage*)
-warn-factor <int> (*prune warn factor*)

The two options **-warn-pct** and **-warn-factor** relate to warnings that may be triggered once the *initial* pruning of a vector is completed. The idea is to issue warnings if initial pruning almost completely destroys a computed vector, as this may be a sign that the pruning parameters should be changed. It depends on the mass remaining after initial pruning whether a warning will be issued. If that mass is less than *warn-pct* or if the number of remaining entries is smaller by a factor *warn-factor* than both the number of entries originally computed *and* the recovery number, in that case, **mcl** will issue a warning.

-warn-pct takes an integer between 0 and 100 as parameter, **-warn-factor** takes a real positive number. They default to something like 30 and 50.0. If you want to see less warnings, decrease *warn-pct* and increase *warn-factor*. Set *warn-factor* to zero if you want no warnings.

-v pruning

Pruning verbosity mode causes **mcl** to emit several statistics related to the pruning process, each of which is described below. Use **-v explain** to get explanatory headers in the output as well (or simply use **-v all**).

IMPLEMENTATION OPTIONS

-sparse <int> (*sparse matrix multiplication threshold*)

This value (by default set to 10) determines when mcl switches to sparse matrix/vector multiplication. For a given column stochastic vector (corresponding with all the neighbours of a given node `v` according to the current mcl iterand) the sum `S` of neighbour counts of all neighbours of `v` is computed, counting duplicates. This is exactly the number of matrix entries involved in the computation of the new column vector for the matrix product. If `S` times *<int>* does not exceed the number of nodes in the graph (equal to both column and row dimension of the matrices used) then a sparse implementation is used. Otherwise an optimized regular implementation is used. Intuitively, this option can be thought of as the estimated overhead per matrix floating point operation incurred by the sparse implementation compared with the regular implementation. MCL uses this estimated overhead to determine which implementation is likely to be quicker. Testing has shown this strategy works very well for graphs of a wide range of sizes, including graphs with up to 3 million nodes and 500 million edges.

NOTE

The effectiveness of this option is influenced by hardware-specific properties such as the CPU L2 cache size. The default value should work reasonably well across a wide variety of scenarios, but it may be possible to squeeze faster run times out of mcl by tuning this parameter to the graphs that are specific for your application domain.

EXAMPLES

The following is an example of label input

```
---8<------8<------8<------8<------8<---
cat hat  0.2
hat bat  0.16
bat cat  1.0
bat bit  0.125
bit fit  0.25
fit hit  0.5
hit bit  0.16
--->8------>8------>8------>8------>8---
```

It can be clustered like this:

```
mcl cathat --abc -o out.cathat
```

The file out.cathat should now like like this

```
---8<------8<------8<------8<------8<---
cat hat bat
bit fit hit
--->8------>8------>8------>8------>8---
```

A few things to note. First, MCL will symmetrize any arrow it finds. If it sees `bat cat 1.0` it will act as if it also saw `cat bat 1.0`. You can explicitly specify `cat bat 1.0`, mcl will in the first parse stage simply end up with duplicate entries. Second, MCL deduplicates repeated edges by taking the one with the maximum value. So,

```
---8<------8<------8<------8<------8<---
cat hat  0.2
hat cat  0.16
hat cat  0.8
--->8------>8------>8------>8------>8---
```

Will result in two arrows `cat-hat` and `hat-cat` both with value 0.8.

APPLICABILITY

mcl will work very well for graphs in which the diameter of the natural clusters is not too large. The presence of many edges between different clusters is not problematic; as long as there is cluster structure, mcl will find it. It is less likely to work well for graphs with clusters (inducing subgraphs) of large diameter, e.g. grid-like graphs derived from Euclidean data. So mcl in its canonical form is certainly not fit for boundary detection or image segmentation. I experimented with a modified mcl and boundary detection in the thesis pointed to below (see **REFERENCES**). This was fun and not entirely unsuccesful, but not something to be pursued further.

mcl likes *undirected input graphs best*, and it really dislikes graphs with node pairs (i,j) for which an arc going from i to j is present and the counter-arc from j to i is absent. Try to make your input graph undirected. Furthermore, mcl interprets edge weights in graphs as similarities. If you are used to working with dissimilarities, you will have to convert those to similarities using some conversion formula. The most important thing is that you feel confident that the similarities are reasonable, i.e. if X is similar to Y with weight 2, and X is similar to Z with weight 200, then this should mean that the similarity of Y (to X) is neglectible compared with the similarity of Z (to X).

mcl is probably not suited for clustering *tree graphs*. This is because mcl works best if there are multiple paths between different nodes in the natural clusters, but in tree graphs there is only one path between any pair of nodes. Trees are too sparse a structure for mcl to work on.

mcl may well be suited for clustering *lattices*. It will depend on the density characteristics of the lattice, and the conditions for success are the same as those for clustering graphs in general: The diameter of the natural clusters should not be too large. **NOTE** when clustering a lattice, you *have* to cluster the underlying undirected graph, and not the directed graph that represents the lattice itself. The reason is that one has to allow mcl (or any other cluster algorithm) to 'look back in time', so to speak. Clustering and directionality bite each other (long discussion omitted).

mcl has a worst-case time complexity O(N*k^2), where N is the number of nodes in the graph, and k is the maximum number of neighbours tracked during computations. k depends on the **-P** and **-S** options. If the **-S** option is used (which is the default setting) then k equals the value corresponding with this option. Typical values for k are in the range 500..1000. The average case is much better than the worst case though, as cluster structure itself has the effect of helping mcl's pruning schemes, certainly if the diameter of natural clusters is not large.

FILES

There are currently no resource nor configuration files. The mcl matrix format is described in the **mcxio(5)** section.

ENVIRONMENT

MCLXIODIGITS

When writing matrices in interchange format, mcl will use this variable (if present) as the precision (number of digits) for printing the fractional part of values.

MCLXIOVERBOSITY

MCL and its sibling applications will usually report about matrix input/output from/to disk. The verbosity level can be regulated via MCLXIOVERBOSITY. These are the levels it can currently be set to.

1 Silent but applications may alter this.
2 Silent and applications can not alter this.
4 Verbose but applications may alter this.
8 Verbose and applications can not alter this (default).

MCLXIOFORMAT

MCL and its sibling applications will by default output matrices in interchange format rather than binary format (cf. **mcxio(5)**). The desired format can be controlled via the variable MCLXIOFORMAT. These are the levels it can currently be set to.

1 Interchange format but applications may alter this.
2 Interchange format and applications can not alter this (default).
4 Binary format but applications may alter this.
8 Binary format and applications can not alter this.

MCLXICFLAGS

If matrices are output in interchange format, by default empty vectors will not be listed. Equivalently (during input time), vectors for which no listing is present are understood to be empty - note that the *presence* of a vector is established using the domain information found in the header part. It is possible to enforce listing of empty vectors by setting bit '1' in the variable MCLXICFLAGS.

MCLXIOUNCHECKED

MCL and its sibling applications will always check a matrix for consistency while it is being read. If this variable is set, the consistency check is omitted. For large graphs the speed up can be considerable. However, if the input graph is not conforming it will likely crash the application that is using it.

DIAGNOSTICS

If **mcl** issues a diagnostic error, it will most likely be because the input matrix could not be parsed succesfully. **mcl** tries to be helpful in describing the kind of parse error. The mcl matrix format is described in the **mcxio(5)** section.

BUGS

No known bugs at this time.

AUTHOR

Stijn van Dongen.

HISTORY/CREDITS

The MCL algorithm was conceived in spring 1996 by the present author. The first implementation of the MCL algorithm followed that spring and summer. It was written in Perl and proved the viability of the algorithm. The implementation described here began its life in autumn 1997. The first versions of the vital matrix library were designed jointly by Stijn van Dongen and Annius Groenink in the period Oktober 1997 - May 1999. The efficient matrix-vector multiplication routine was written by Annius. This routine is without significant changes still one of the cornerstones of this MCL implementation.

Since May 1999 all MCL libraries have seen much development and redesign by the present author. Matrix-matrix multiplication has been rewritten several times to take full advantage of the sparseness properties of the stochastic matrices brought forth by the MCL algorithm. This mostly concerns the issue of pruning – removal of small elements in a stochastic column in order to keep matrices sparse.

Very instructive was that around April 2001 Rob Koopman pointed out that selecting the k largest elements out of a collection of n is best done using a min-heap. This was the key to the second major rewrite (now counting three) of the MCL pruning schemes, resulting in much faster code, generally producing a more accurate computation of the MCL process.

In May 2001 Anton Enright initiated the parallellization of the **mcl** code and threaded inflation. From this example, Stijn threaded expansion. This was great, as the MCL data structures and operands (normal matrix multiplication and Hadamard multiplication) just beg for parallellization.

Onwards. The January 2003 03-010 release introduced support for sparsely enumerated (i.e. indices need not be sequential) graphs and matrices, the result of a major overhaul of the matrix library and most higher layers. Conceptually, the library now sees matrices as infinite quadrants of which only finite subsections happen to have nonzero entries.

The June 2003 03-154 release introduced unix-type pipelines for clustering, including the BLAST parser mcxdeblast and the mclblastline script. The April 2004 04-105 release revived binary format, which has been a first class citizen every since.

With the March 2005 05-090 release mcxsubs finally acquired a sane specification syntax. The November 2005 05-314 release brought the ability to stream label input directly into mcl. The subsequent release introduced a transformation language shared by various mcl siblings that allows arbitrary progressions of transformations to be applied to either stream values or matrix values.

Joost van Baal set up the mcl CVS tree and packaged mcl for Debian GNU/Linux. He completely autotooled the sources, so much so that at first I found it hard to find them back amidst bootstrap, aclocal.m4, depcomp, and other beauties.

Jan van der Steen shared his elegant mempool code. Philip Lijnzaad gave useful comments. Philip, Shawn Hoon, Abel Ureta-Vidal, and Martin Mokrejs sent helpful bug reports.

Abel Ureta-Vidal and Dinakarpandian Deendayal commented on and contributed to mcxdeblast and mcxassemble.

Tim Hughes contributed several good bug reports for mcxassemble, mcxdeblast and zoem (a workhorse for **clm format**).

SEE ALSO

mclfaq(7) - Frequently Asked Questions.

mcxio(5) - a description of the mcl matrix format.

There are many more utilities. Consult **mclfamily(7)** for an overview of and links to all the documentation and the utilities in the mcl family.

minimcl is a 200-line perl implementation of mcl. It is shipped in the mcl distribution and can be found online at http://micans.org/mcl.

mcl's home at http://micans.org/mcl/.

REFERENCES

[1] Stijn van Dongen, *Graph Clustering by Flow Simulation.* PhD thesis, University of Utrecht, May 2000.
http://www.library.uu.nl/digiarchief/dip/diss/1895620/inhoud.htm

[2] Stijn van Dongen, *Graph Clustering Via a Discrete Uncoupling Process*, SIAM Journal on Matrix Analysis and Applications, 30(1):121-141, 2008.
http://link.aip.org/link/?SJMAEL/30/121/1

[3] Stijn van Dongen. *A cluster algorithm for graphs.* Technical Report INS-R0010, National Research Institute for Mathematics and Computer Science in the Netherlands, Amsterdam, May 2000.
http://www.cwi.nl/ftp/CWIreports/INS/INS-R0010.ps.Z

[4] Stijn van Dongen. *A stochastic uncoupling process for graphs.* Technical Report INS-R0011, National Research Institute for Mathematics and Computer Science in the Netherlands, Amsterdam, May 2000.
http://www.cwi.nl/ftp/CWIreports/INS/INS-R0011.ps.Z

[5] Stijn van Dongen. *Performance criteria for graph clustering and Markov cluster experiments*. Technical Report INS-R0012, National Research Institute for Mathematics and Computer Science in the Netherlands, Amsterdam, May 2000.
http://www.cwi.nl/ftp/CWIreports/INS/INS-R0012.ps.Z

[6] Enright A.J., Van Dongen S., Ouzounis C.A. *An efficient algorithm for large-scale detection of protein families*, Nucleic Acids Research 30(7):1575-1584 (2002).

NOTES

This page was generated from **ZOEM** manual macros, http://micans.org/zoem. Both html and roff pages can be created from the same source without having to bother with all the usual conversion problems, while keeping some level of sophistication in the typesetting.