Nodes and edges with METAPOST: The MetaGraph environment

Federico García De Castro

1 Introduction

The aim of this article is to present MetaGraph, a set of METAPOST macros and utilities developed over the last couple of months as an open-ended environment for drawing *graphs* (in the sense of "nodes and edges"), and intended to complement external graph analysis engines with the versatility of programmatic formatting.

After a quick glance at MetaGraph's capabilities through three demo graphs, the sections below offer a general description of the system, highlighting the *data vs. procedure* approach that makes it different from the plotting routines typically available in those external engines — with a special mention of Ti*k*Z — and offering a general 'feel' for what this approach entails and permits.

All graph-related techniques and terms mentioned here are used simply for illustration purposes, and the details of what they mean in graph theory or analysis do not matter much. What the 'degree' of a node illustrates in this or that example could just as well have been illustrated by its '*k*-core number', its various 'centrality' measures, or any such node attribute — I will therefore not be discussing these concepts in any depth. Similarly, I will not go into much detail regarding METAPOST's general syntax — knowledge of METAPOST is surely a good asset for using MetaGraph, but I don't think it's a pre-requisite: the operations shown here should be good base analogs for anyone potentially interested in using the system, even without prior METAPOST experience. I am happy to share the source code.

1.1 Three demo graphs

The graph in Figure 1 is a so-called 'random geometric' graph with 200 nodes and 860 edges, drawn with *a*) two kinds of node marker (● and ○) and *b*) "lassos", that highlight two particular features of the graph (resulting from two particular graph analysis techniques).[1]

Figure 2 is a graph made from a much larger set of data, but from an abstract point of view it is essentially the same kind of object: a set of name-less nodes and direction-less edges.[2] The figure itself is a much plainer representation of the graph — just black node markers and grey edges — but there is nothing to prevent the kind of lassoing, conditional formatting, or other graphic treatment that was done on the first graph. The main reason to include Figure 2 here was to test the limits of MetaGraph; as it turns

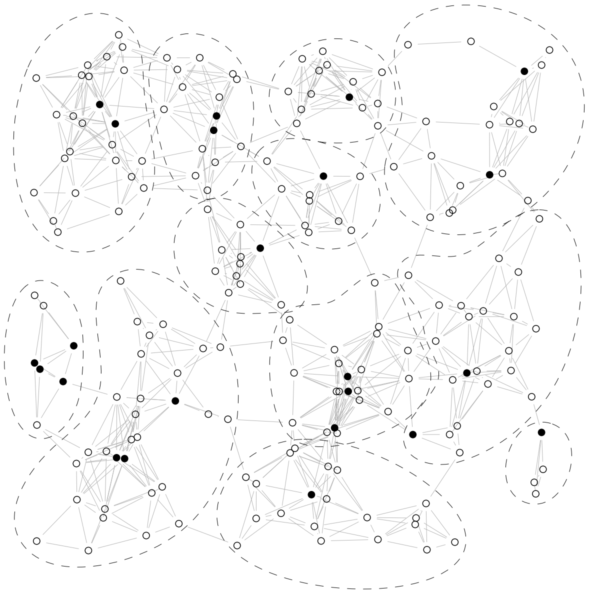

Figure 1: A random geometric graph, 200 nodes and 860 edges

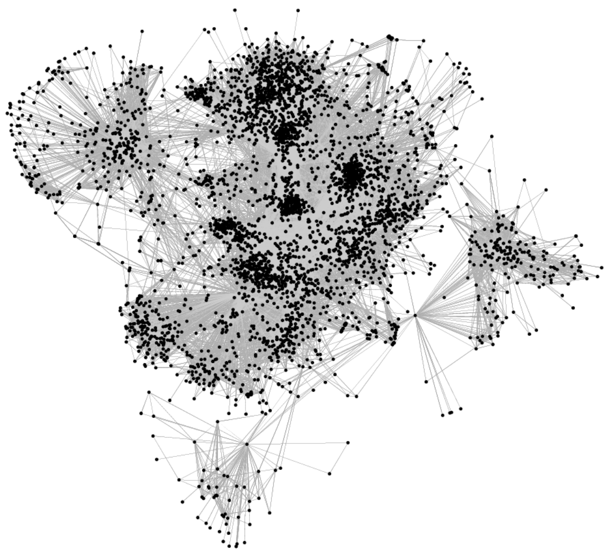

Figure 2: A region of Facebook, 4,039 nodes and 88,234 edges

[1] For graph theory folks: The ●-nodes are nodes with 'revcore dependency' equal to 0 (meaning that they are local centers of density according to 'reverse core decomposition'); the groups of nodes lassoed in the figure are the node communities yielded by the 'Louvain community detection' algorithm. Both things were computed in a graph-analysis engine, and fed to MetaGraph in a 'data file', as will be explained in more detail below.

[2] These two graphs come from the documentation of the Python library 'Networkx' at `networkx.org/nx-guides/content/exploratory_notebooks/facebook_notebook.html` and `networkx.org/documentation/stable/auto_examples/drawing/plot_random_geometric_graph.html`.

doi.org/10.47397/tb/45-1/tb139garcia-metagraph

out, even its 88,234 edges are far from exhausting METAPOST's capacity (see section 3). The limit on how large a graph MetaGraph can deal with is likely to be practical rather than computational.

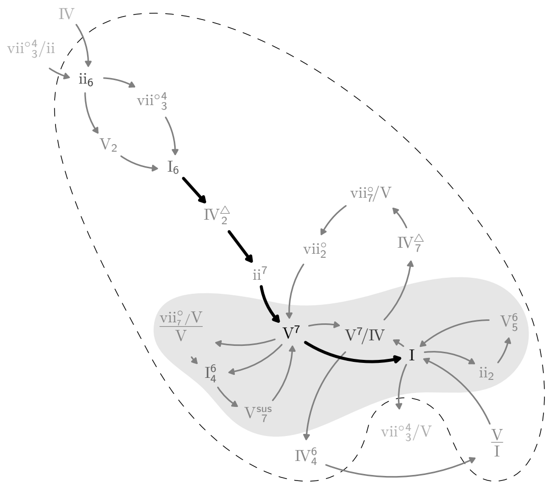
IV
$\text{vii}^{\circ 4}_{3}/\text{ii}$
ii_6
$\text{vii}^{\circ 4}_{3}$
V_2
I_6
$\text{IV}^{\triangle}_{2}$
$\text{vii}^{\circ}_{7}/\text{V}$
vii°_{2}
$\text{IV}^{\triangle}_{7}$
ii^7
$\frac{\text{vii}^{\circ}_{7}/\text{V}}{\text{V}}$
V^7
V^7/IV
V^6_5
I
I^6_4
ii_2
$\text{V}^{\text{sus}}_{7}$
$\text{vii}^{\circ 4}_{3}/\text{V}$
$\frac{\text{V}}{\text{I}}$
IV^6_4

Figure 3: A directed graph of the harmonies of the first prelude in J.S. Bach's *The Well-Tempered Clavier*

The third demo graph is different in a couple of deeper senses. It is a 'directed' graph — hence the arrows in Figure 3 — and its nodes have names — hence the node labels instead of markers. The visualization in the figure features *a*) both a lasso and a shaded region — for METAPOST the two things are not too different, the former resulting from '`draw dashed evenly`', the latter from '`fill`' — and *b*) conditional formatting on the nodes and edges, according to node 'degree' and edge 'weight' (details in section 2.4).

It is this kind of graph that I have been working on in the context of music-theory research, and it is the need for chord-ciphers such as $\frac{\text{vii}^{\circ}_{7}/\text{V}}{\text{V}}$, annoying outside of TEX, that led me to *a*) explore the possibility of writing some useful macros; *b*) realize how adequate METAPOST is for these matters; and *c*) share the news. I imagine the truly easy handling of nodes as LATEX expressions has wider appeal ('H_2SO_4', etc.).

1.2 MetaGraph, Ti*k*Z, etc.

To be sure, most relevant external environments can handle LATEX, 'importing' it or its output into their workflow. In particular, Ti*k*Z (which handles TEX natively, of course) has a truly sophisticated library for graphs, which includes even algorithms that are not present in all graph analysis engines — as well as facilities to implement new ones. Ti*k*Z offers options for the style, placement, coloring, and even animation of nodes, as well as for (several) ways of connecting them with edges. In addition, Ti*k*Z is well integrated in the graph-analysis landscape, and most engines have a backend to export their graphs as Ti*k*Z code, just as they can typically export them as matplotlib plots.

There is, however, little overlap between Ti*k*Z and MetaGraph — in fact, just as little as there is between MetaGraph and matplotlib. MetaGraph is not intended to be a self-contained unit implementing a comprehensive syntax for every possible node- and edge-drawing need and option. If anything, it pursues the opposite: as open-ended an environment as possible, where the graph (its data, essentially) is little more than 'set up' for later straightforward drawing through METAPOST. The exact nature of this approach, its possible benefits and utility, and its difference with Ti*k*Z, will perhaps be clearer after reading or even only surveying the sections below.

1.3 What is MetaGraph?

1.3.1 A set of METAPOST macros

In the strictest sense, MetaGraph is simply a set of METAPOST macros: high-level shortcuts that expand into the plain METAPOST constructions that draw edges, add node labels, etc.

For example, the `allnodes` macro expands to '`0 upto last_node`', which one can use freely to loop, '`for node = allnodes`'. Another macro, `addlabel`, expands to

`addto currentpicture also label`⟨*node-id*⟩

where `label` in turn stands for '`nlabels`⟨*node-id*⟩ `shifted pos`⟨*node-id*⟩' — and so on: `pos` is itself a macro (more on this in section 2.1).

There are also more complicated routines. When drawing an edge (`arrowedge` or `lineedge`), the system checks on the labels of the two nodes involved, so that the edge is drawn starting and ending on the corresponding intersection points, depending on various values such as `labelpadding`, `edgeangle`, etc. There is a family of utilities — `leftoflabel`⟨*node*⟩, `belowrightoflabel`⟨*node*⟩, `abovelabel`⟨*node*⟩, and others — that return the coordinates of the requested point, so that they can be used, for example, in the drawing of lassos.

1.3.2 A graph-drawing *system*

Through macros like these — collected in the METAPOST file `metagraph.mp` — MetaGraph acts as an interface between the actual node and edge data and the METAPOST operations that are most commonly useful to draw the actual graph out of those data. In other words: *given the node and edge data*, there are macros in MetaGraph that provide high-level, graph-oriented utilities to have METAPOST produce the images.

But these 'node and edge data' are expected to come from somewhere else — typically an external graph analysis engine — and they must follow certain conventions in order to be understood by `metagraph.mp`. In this wider sense, MetaGraph is actually a *system* for drawing graphs, consisting, at the time of writing, of *a*) the macros, and *b*) the conventions that need to be followed to provide the raw data of the graph. Eventually, MetaGraph should/will also include *c*) documentation, and *d*) backends for the most common graph-analysis engines.[3]

1.3.3 Dependencies and work flow

One of the good things about METAPOST (over its inspiration, METAFONT) is that it handles TEX natively. It may still give some installation/configuration trouble, since it needs to be pointed to the actual TEX engine used (plain? LATEX? other?); but it is a primitive feature of METAPOST.

One needs also communication in the other direction: from METAPOST into TEX. Since METAPOST produces generic image files (PNG or SVG in addition to PostScript), this is covered by the existing methods to import images into TEX documents.

Of note in this connection is also LuaTEX/LuaLATEX, and its METAPOST library/package luamplib, that takes full care of the communication in both directions (with luamplib, a METAPOST picture is just a TEX box). If one uses Overleaf, then there is no hassle in ensuring that the different tools and formats are well installed, mutually aware, etc.

1.3.4 Under construction

It must be said that MetaGraph is under construction. It is fairly operational — the graphs in these pages were all created with its existing routines — but there are points of syntax still undergoing improvement, a lingering low-levelness, and a chance that further routines may be identified and implemented as I myself become familiar with its capabilities. (The lassoing routines, for example, were entirely developed during and because of the writing of this article — they were only an intuition before I ran into the actual occasion to develop them.) More damaging yet, I have yet to compile a complete, even complete enough, reference manual.

MetaGraph will sooner or later make its way to CTAN, and, in the form of backends, to Networkx and other external graph tools. In the meantime,

[3] At present, there exists the one I wrote for the combination of Networkx and pandas that I use for graph analysis in Python.

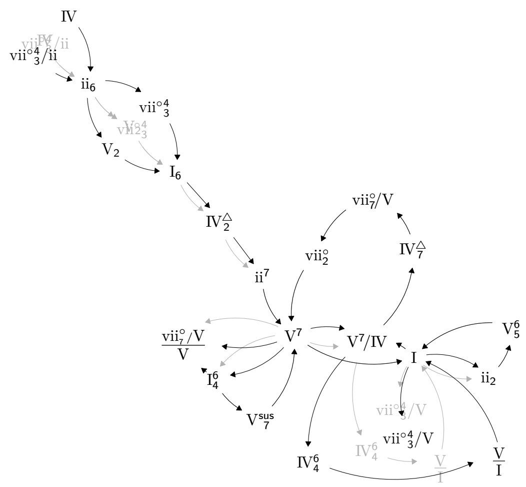
IV
$\mathrm{vii}^{\circ 4}_{3}/\mathrm{ii}$
ii_6
$\mathrm{vii}^{\circ 4}_{3}$
V_2
I_6
$\mathrm{IV}^{\triangle}_{2}$
$\mathrm{vii}^{\circ}_{7}/\mathrm{V}$
vii°_{2}
$\mathrm{IV}^{\triangle}_{7}$
ii^7
$\frac{\mathrm{vii}^{\circ}_{7}/\mathrm{V}}{\mathrm{V}}$
V^7
V^7/IV
I
V^6_5
ii_2
I^6_4
$\mathrm{V}^{\mathrm{sus}}_{7}$
$\mathrm{vii}^{\circ 4}_{3}/\mathrm{V}$
IV^6_4
$\frac{\mathrm{V}}{\mathrm{I}}$

Figure 4: Tweaking of the Figure 3 graph of Bach's prelude

if there is any interest, I am happy to share both the macro definitions and the source code for the illustrations here.

2 Data vs. drawing

The key fact about MetaGraph as a system is that it keeps a complete separation between the graph data and the drawing operations. The first line of code in a MetaGraph graph is usually

```
  input metagraph;
```

(the library of METAPOST macros), while the second is, for example,

```
  input rgdata;
```

(the data file for the random graph of Figure 1).

The two files are completely independent; they are each useless by themselves, but they know exactly nothing about each other.

This way of proceeding preserves two things: *a*) a general programming environment, where the data is assigned for its own sake, and can be used in any way by any METAPOST routine; and, as a result, *b*) direct access to individual graph nodes and edges, for manual or programmatic manipulation, inside or outside other procedures, in the same figure or in a different one.

Figure 4, for example, shows the graph of Bach's prelude (the one in Figure 3), laying a 'tweaked' version in black ink over a lighter-shade layout (yielded by the Fruchterman-Reingold algorithm, one of the 'force-directed' algorithms for laying out graphs). The tweaks include the nudging of certain nodes, to avoid collisions present in the original layout in some cases — in others simply to accommodate the lasso of Figure 3. Some edges are also tweaked. Curved arrows are usually good for directed graphs — they make bi-directed edges easier to see — but the default outgoing angle, calculated blindly with respect to the source node, often creates less-than-satisfying layouts.

Theoretically, these tweaks are of course less than crucial; but they provide a good illustration of the benefit of direct access to individual nodes and edges. Three kinds of commands were used to make the tweaks in Figure 4 (only one instance of each is listed):

```
npos[IV] := % avoid collision
    npos[IV] rotated -3 + (.05, .05);
set_angle((IVj2, ii7), 0); % straight arrow
flipangle(IV, ii6); % flip the arrow
```

This kind of manipulation is much less straightforward in a system where both the data of nodes and edges and the global graphic options (arrows on/off, edge angle such and such, etc.) are issued in the same line.[4]

2.1 Nodes, indices, and arrays

What we have been calling the 'node data' is simply a series of METAPOST arrays. A 'node', for MetaGraph, is just the index number that points to the node's place in those arrays (naturally it is the same in all of them).

In other words, the node's index is the ⟨*node-id*⟩ in expressions like `addlabel`⟨*node-id*⟩, for example. As mentioned, this particular macro expands to the METAPOST line '`addto currentpicture also label`⟨*node-id*⟩', passing ⟨*node-id*⟩ along to `label`. The latter will pass it on in turn as `pos`⟨*node-id*⟩ and `nlabels`⟨*node-id*⟩:

- The data file includes the position of the nodes in the array `npos`⟨*node-id*⟩. Graph-analysis engines typically issue position information without thinking of a particular point size (usually normalizing to the first quadrant, or to the unit circle, etc.), and therefore the values in `npos` need to be scaled; this is what `pos`⟨*node-id*⟩ does.
- Unlike node positions, node labels are not required by MetaGraph. If a graph does have node labels, its data file will have created the array `nlabels`, where MetaGraph will be able to look up each node's LaTeX expression. In label-less graphs, `nlabels` does not exist, and MetaGraph will simply supply the default node image (or whatever the user may define for it).

All throughout the process, ⟨*node-id*⟩ picks out the information for the particular node at hand from each of the data arrays.

[4] One would have to issue three subgraphs: one with the straight edges, one with the clockwise edges, and one with the counterclockwise edges.

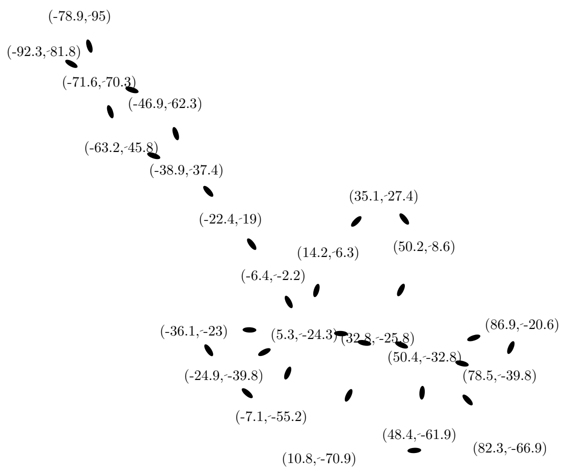

Figure 5: A funny view of the Bach prelude's graph, with the (rounded) raw node positions as 'labels', and blobs — rotated in the direction from one node to the other — in the midpoint of each edge.

2.2 General programming environment

The preceding is the fate of ⟨*node-id*⟩ through the `addlabel` process; but rather than illustrating the workings of that particular macro, the point here is to stress that we are performing general, open-ended programming on raw values with no intrinsic semantics. Adding the label will be the most common use for the node position value; but *at all times* this is just a `pair` variable like any other, and it can be used as such. We can use it — as in Figure 5 — as the label itself, or find the mid-point between two of them, or find out, for whatever purpose, `if angle(npos`⟨*node-id*⟩`) > ctcl_angle`.

2.3 The lassos

Interestingly, the general-purpose nature of METAPOST as a programming language makes MetaGraph a suitable environment in which to implement graph-theoretical techniques. It is relatively easy, for example, to extract a graph's 'line-graph' (a version of the graph whose nodes are the original graph's edges, connected iff they originally share a node), or perform '*k*-means' or 'DBSCAN' clustering. But these techniques are, after all, likely to be available in whatever graph-analysis engine one is using in connection with MetaGraph.

This is *not* the case with lassoing. Figure 6 zooms in on one of the communities (just SE of the origin) of the random graph in Figure 1. This particular lasso (the fifth place in an array of METAPOST paths created for the purpose, called `comms`) was produced by the following code:

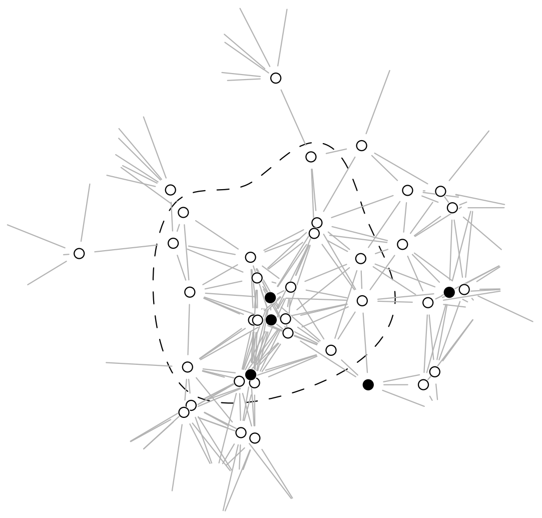
Figure 6: A close-up of one of the communities in the random graph of Figure 1

```
comms[5] = 1/2[pos159, pos152]..tension 1.5
    ..1/2[pos129, pos40]..tension 1.3
    ..1/2[pos103, pos158]
    ..1/2[pos137, pos90]
    ..1/2[pos22, pos14]
    ..1/2[pos163, pos4]
    ..abovelabel168{left}
    ..1/2[pos152, pos168]
    ..cycle;
```

Except for '`abovelabel`' (a MetaGraph macro that finds the point above the label of a given node), this is all plain METAPOST syntax:

- The '`..`' connector creates a (Bézier) curved path between two points. ('`--`' would create a straight line.)
- The `tension` t modifier (with $t = 1$ by default) is one of the ways to control the Bézier curves thus created. (Another is '`controls` c_1 `and` c_2', to specify explicitly the two control points.)
- The convenient $\frac{a}{b}[p_1, p_2]$ construction finds the coordinates of the point that lies $\frac{a}{b}$ of the way from p_1 to p_2.
- '`{left}`' tells METAPOST that the path should travel left at that particular point. (Naturally, one can direct a path `{right}` instead, or `{up}`, or `{down}`, or indeed any `{`⟨*custom vector*⟩`}`; if one wants a particular angle θ, '`dir(`θ`)`' provides the corresponding vector.)

This path (and the others) was designed by visual inspection and trial-and-error. From a run of the 'Louvain community detection algorithm' in Networkx I knew the sets of nodes (i.e., node indices) in each community, and I was looking at a separate version of the graph with the node indices as labels, so that I could locate each community and design its lasso. There is essentially no algorithmic way of lassoing arbitrary sets of nodes, but direct access to node information for varied uses makes it possible to generate paths quickly and robustly.

Incidentally, the clipping of the graph to zoom in on this particular region was also made through node information: for Figure 6, I instructed MetaGraph to include only those nodes and edges that are less than 50 points removed from the center of the lasso (calculated ahead of time as '`lassoctr`'):

```
for edge = alledges:
  if (length(pos[source(edge)]-lassoctr) < 50)
  or (length(pos[target(edge)]-lassoctr) < 50):
    lineedge(edge) withcolor .7white;
  fi;
endfor;
for node = allnodes:
  if length(pos[node]-lassoctr) < 50:
    addlabel[node];
  fi;
endfor;
```

2.4 Node (and edge) attributes

So far we have only referred to node positions and node labels as part of the data that MetaGraph expects to find in the graph data file. Other pieces of information are also required contents of the data file; we will discuss those in section 2.6. Here we shall deal with the possibility of adding and manipulating custom data.

Many attributes of nodes and edges are often relevant for the way a graph is represented graphically (since they are often what needs to be shown). This includes both intrinsic attributes like weights, degrees, etc., and information resulting from global graph analysis — things like k-core and -truss numbers, various 'centrality' measures, etc.[5]

Just like positions and labels, all these attributes are raw data for MetaGraph. But while the arrays of positions and labels are node-specific, one-by-one arrays of n values (n being the number of nodes), most other attributes are encoded in the data file as arrays of *lists* — one list for each value v of the attribute; paraphrasing, lists like "the nodes of attribute *foo* equal to value v are: these and these".

This is much more efficient and straightforward to use than node-by-node arrays, since most metrics

[5] The shaded region in Figure 3 (page 118), for example, is the '3-truss' of the graph, where every edge is part of at least $(3-2)$ triangles, and the lasso its '2-core', where every node has at least 2 edges.

The 'degree' of a node is the number of edges incident upon it; edge 'weights' usually encode empirical observations of a kind or another — in the harmonic-behavior graphs like that of Bach's prelude, they represent the frequency of the progression between two chord-nodes.

group nodes and edges, rather than having single values for each: rather than looping over nodes and checking for their *foo* value, the program (or the user) can loop over the list of *foo*-valued nodes.

The full specification of an attribute `foo` in MetaGraph consists of:

- The pluralized `foos`, which holds a list of the possible values the attribute takes.
- Single-value variables `foo_min` and `foo_max`.
- Explicit lists for each value, `foo_values`⟨*value*⟩, whose contents is a list of the corresponding indices.
- A macro `foo` that expands these lists.[6]
- A prefix: all of the above are actually `nfoo` or `efoo`, according to whether they are node or edge attributes.

For example, the attribute 'frequency' of the nodes in the graph of Bach's prelude is the number of occurrences of each particular chord in the piece. It can be represented as follows in the graph's data file:

```
def nfreqs = 1, 2, 4, 5 enddef;
def nfreq = scantokens nfreq_values enddef;
string nfreq_values[]; % declares array
nfreq_values[1] =
    "ii2, V65, ii6, V2, viid43, IVj7, sivd7,
    viid2, sivd7oV, sid43, sivd43";
nfreq_values[2] =
    "IV, I6, ii6, IVj2, ii7, I7, I64, Vsus7,
    IV64, VoI";
nfreq_values[4] = "I";
nfreq_values[5] = "V7";
nfreq_min = 1; nfreq_max = 5;
```

Note that the lists are not given in terms of node index numbers, but rather expressions that resemble the labels of the nodes: '`ii2`', etc. This will be addressed in the next section.

Armed with these lists, conditional formatting based on a given attribute is straightforward. The original view of the graph (Figure 3), for example, makes the grey level and the width of the edges depend on the attribute `efreq`:

```
for freq = efreqs: % freq values present
  for edge = efreq[freq]: % edges of each freq
    arrowedge(edge)
      withpen pencircle scaled ((freq/5)*mm)
      withcolor (.5*(efreq_max-freq))*white;
endfor; endfor;
```

The shading of the nodes, in turn, is a function of another node attribute present in the data file — the node degree:

```
for dg = ndegrees: % degree values present
  for node = ndegree[dg]: % nodes of each dg
    addlabel[node]
      withcolor (.8 - dg/ndegree_max)*white;
  endfor;
endfor;
```

Being raw data, the attributes can just as well be used independently of the graph itself — for example, to produce attribute distribution diagrams, common in graph analysis. Here is the relevant loop for the node degrees of the random graph of Figure 1:

```
pickup penrazor xscaled 6pt;
for dg = ndegrees:
    count := 0;
    for nodes = ndegree[dg]:
        count := count + 1;
    endfor;
    draw (6*dg, 0) -- (6*dg, 2*count);
endfor;
```

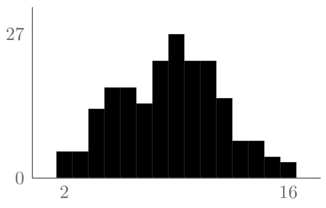

Figure 7: Degree distribution of the graph in Figure 1

2.5 Nodes, labels, and aliases

We have seen that the MetaGraph's ⟨*node-id*⟩s are numbers (section 2.1). But nothing prevents one from assigning these numbers to a series of named variables, which then function essentially as node aliases.

This is particularly relevant in graphs with labeled nodes. The data file of such a graph should contain ASCII-friendly versions of the node labels, as variable names equaling each node's index. That is the case with the graph of the Bach prelude, whose data file creates these aliases as its first order of business:

```
% Nodes
I = 0;
IV = 1;
...
viid43 = 16;
viid2 = 17;
...
```

As we have seen, even the data file uses these variables (rather than the indices) to define the lists of node attributes. These aliases are in fact *entirely* equivalent to the indices; both the program and — more to the point — the user can use them to operate on the nodes (flipping or setting the angle of a

[6] Lists are not a METAPOST data type; they are implemented as string variables, using the primitive `scantokens` to process them; `foo` takes care of this.

curved edge or nudging the position of the nodes, as we did in Figure 4, or manually drawing edges beyond those included in the data file, etc.) without ever needing to care about the otherwise meaningless index numbers.

It is quite fortunate then that METAPOST's variable name conventions allow for monstrosities like `viid43ofii`, or possibly `H2SO4`, and of course benign things like `a`. (The latter is not so benign after all; before I knew better, I was *repeatedly* perplexed by some strange graph-drawing behaviors, only eventually to find that a '`for i`' loop had been clashing with an 'i' that was one of my nodes.)

As long as it starts with a letter, virtually any alphanumeric string is a valid METAPOST variable. It may seem inconvenient to have to refer to $\frac{\mathrm{vii}^{\circ}_{7}/\mathrm{V}}{\mathrm{V}}$ through the all-ASCII name `viid7ofVoV`. But the concession does not come at the MetaGraph stage: already in the graph-analysis engine, if one needs to access nodes individually — to set or get their attributes, for example, or to read out a list of nodes resulting from some clustering operation — then code-friendly names are all but required. Since the engine will then already know these aliases, it can easily add them as variable assignments in the data file. The same names will be available to use at the drawing stage with MetaGraph if they all start with a letter.

2.6 The data file

What attributes to include in a MetaGraph data file is almost entirely discretionary; only a few pieces of information are required. One of them, as mentioned, is the node positions, that go in the `npos` array; another is the `last_node` index, necessary for MetaGraph to loop over all the nodes.

Not mentioned so far is an addition array required by MetaGraph: the list of each node's (outgoing) neighbors, `nodes_to`.

It may not be immediately obvious why this should be required. On the one hand I can report that I have often found myself needing that information for particular graph drawings; on another, it opens up the possibility of having MetaGraph extract subgraphs — although this involves multiple levels of nested loops, and at some point it is better to have the external graph-analysis engine do the work and produce a separate data file. Mostly it has to do with implementing edges in an efficient way, as we will see in the next section.

A minimal data file will then be something like this (coming from `rgdata.mp`, the data file for the random graph in Figure 1):

```
last_node = 199;
% npos (pair)
pair npos[]; i_ := -1;
for value = (0.585, 0.229), (0.722, 0.533),
            ...
            (0.398, 0.516), (0.056, 0.328):
    npos[incr i_] = value;
endfor;
% nodes_to (string)
string nodes_to[];
    nodes_to[4] = "1";
    nodes_to[7] = "5";
    ...
    nodes_to[199] = "167, 13, 81, 50, 23";
```

Then the file may go on to set up the non-required attributes. For example, the node degree attribute in `rgdata.mp` is given by:

```
% ndegree (numeric)
string ndegree_values[]; % declares the array
def ndegrees = 2, 3, 4, 5, 6, 7, 8, 9,
               10, 11, 12, 13, 14, 15, 16
  enddef;
  ndegree_values[2] = "25, 76, 100, 184, 196";
  ndegree_values[3] = "24, 65, 92, 171, 181";
  ...
  ndegree_values[16] = "131, 169, 182";
ndegree_min = 2; ndegree_max = 16;
def ndegree = scantokens ndegree_values enddef;
```

2.7 The edges

We have not dealt much with edges so far. Just as a node in MetaGraph is a number, a MetaGraph edge is a *pair* of numbers: the indices of the source and target nodes. As a graphical, coordinate-based language, METAPOST has a full infrastructure for tuples of 2 numbers — its data type `pair`. Edges are not coordinate pairs, of course, but they are METAPOST `pair`s.

The first implementation of MetaGraph treated the edges of a graph as an array of such pairs, each given an ⟨*edge-id*⟩ number. Edge attributes were analogous to node attributes: lists of ⟨*edge-id*⟩ numbers held in string variable arrays.

But the experiments with the Facebook graph in Figure 2 revealed something interesting, having to do with the assignment of an index number to each edge. That is: to each of all *88,234* of them... METAPOST users will know that this goes over the language's 4,096 arithmetic limit, but this is not the issue: there are ways to get around it (see section 3). The problem was that the program would take way, way too long — I could not bear to let it finish a single time — just reading the data file...

This seemed to doom the whole idea of MetaGraph, or at least limit it to relatively small graphs with a manageable number of edges. But it is easy to see that the '`edges`' array is redundant, especially as we have a list of neighbors for each node (see section 2.6). Storing the edges of the Facebook graph in this way entails at most 4,039 assignments (actually somewhat less, since not all nodes have outgoing neighbors). This is a significant reduction — enough to make the program run smoothly. The change makes it a little more complicated to loop over the edges, which now means looping over each node and then again over its `nodes_to`. But the nuisance is small enough, and the macro `alledges` implements the loop, even taking care of neighbor-less nodes (which would normally make the loop break).

Edge attributes are still possible, and they are still lists, analogous to those for nodes ("edges of attribute *foo* equal to value v are: these and these"), only holding `pair` values instead of single numbers. For example, this is the list of edges of frequency 2 in the Bach prelude graph:

```
efreq_values[2] = "(ii7, V7), (IVj2, ii7),
    (I6, IVj2), (V7, I)";
```

One can still loop over these lists and format the edges conditionally — the dark arrows in Figure 3 were ultimately a loop over this very list. Furthermore, nothing prevents one from creating new lists of edge attributes, or indeed new edges, as needed, in real time (i.e., outside the data file).

3 How much is too much?

Larger and larger graphs will of course exceed the system's capacity at some point. But even the 4,039 nodes and 88,234 edges of Figure 2 are far from exhausting METAPOST's (much less LuaTEX's) memory, stack sizes, etc. The graph, in fact, is processed by MetaGraph in a noticeably shorter time than matplotlib takes to draw it (and then again to display it, zoom over it, etc).

It is still a lucky coincidence that that particular graph has 4,039 nodes — dangerously close but still shy of METAPOST's arithmetic bound of 4,096. (This bound, you may recall, is not a matter of physical capacity, but a fundamental feature of the language: there simply exists no $n > 4096$ in METAPOST.)

This issue is not particularly hard to get around. The subscripts of a METAPOST array do not have to be natural numbers, and we could assign the attributes of a node of index i at position $\frac{i}{2}$ of the arrays — where in 'normal' circumstances we assign just i. This would give us up to 8,192 nodes. The same capacity would be achieved by wrapping around and using negative indices as well. Thus, indices of the form, say, $\pm i/10$, would give us 81,920 possible nodes.

Still, in an earlier draft of this paper, where the Facebook graph came after many other figures, the processing of the document *did* overflow Overleaf's free-version compile time. (The paid version, 12× faster, has no problem.) What can be done in those cases?

The solution is to produce a PNG through an external run of METAPOST, and `\includegraphics` it in the document. As a final example, the file that produced Figure 2 is reproduced below in its entirety. The first couple of lines are the ones that make METAPOST produce a PNG file ('`fbtopng-1.png`'); the rest is plain MetaGraph.

```
outputformat := "png";
outputtemplate := "%j-%c.png";

input metagraph;
input fbdata;

scale = 180;

beginfig(1);
  pickup pencircle scaled .1pt;
  for node = allnodes:
    for tgt = nodes_to[node]:
        draw pos[node] -- pos[tgt]
          withcolor .8white;
    endfor;
  endfor;

  addnodes;
endfig;
end.
```

◇ Federico García De Castro
Professor of Composition and Theory
EAFIT University
Medellín, Colombia
fgarciac1 (at) eafit dot edu dot co