Proposal to Improve Visualizations in NetworkX by Ben Edwards

Ben Edwards

April 3, 2011

1 Synopsis

NetworkX is a powerful tool for the analysis of complex networks; however, while its library of algorithms is very expansive, NetworkX has limited visualization capacities. Currently the main avenue for visualization is the matplotlib library, but several features (such as correctly drawn arrows, easily achieved variable node shape, and interactive node positioning) are missing. Additionally, many of the more striking results require a good working knowledge of many of matplotlib's features. This project seeks to not only improve the matplotlib drawing interface, but to refactor the code in such a way that new drawing interfaces could be easily added NetworkX. Because a number of open source projects exist for visualizing and manipulating graphs and data (Gephi, cytoscape, mayavi), this project would seek to create a standard way for NetworkX to communicate with outside visualization methods, rather than reinvent an already robust code base.

2 Project Details

2.1 Current Drawing Interface

Currently NetworkX, does not support a standard way for graphs to be visualized using disparate drawing interfaces. Each of the currently supported drawing interfaces (dot, graphviz, and matplotlib) are implemented uniquely (though similarly), with different arguments for various drawing functions. While the dot format does not actually render graphs, rather it just creates a dot file, PyGraphViz and matplotlib can be used to render graphs. Unfortunately the `view_pygraphviz` function and `draw_networkx` function, take different parameters, and generally produce different results. Moreover, neither of these methods allow for a simple method to have a rendered graph change along with topological graph changes.

2.2 Visualization Engines

A portion of this project would be to rectify this through the creation of several classes and small (but completely backwards compatible) changes to the base NetworkX graph class. First it would require the construction of a `VisualizationEngine` class. This would be a wrapper class, built for the purpose of providing a standard interface for users wishing to display their graphs.

This class could be created to monitor the desired appearance of the graph to be visualized, parsing user input, and allowing the user to set a number of node attributes, including, but not limited to

Nodes	Edges	Labels (Nodes and Edges
Position	Color	Font
Color	Stipple (Style)	Size
Size	Thickness	Color
Shape	Arrows	Alignment
Border	Arrow Style	Position
Border Color		
Border Size		

Each of the above would have a number of options for the user, including sensible defaults. For instance color might be passed as either numeric values, strings (hexadecimal or html colors), iterative of length 3 or 4 (RGB or RGBA values), or functions which return one of these types, or lists or dictionaries of the above types to map onto a node. The visualization engine class could then parse the input into a standard format, for example a dictionary of RGBA tuples keyed by node might be sensible. The engine would simply then pass each parsed value onto the appropriate *specific* visualization engine.

Specific visualization engines could then be written for specific backends, and easily incorporated into current code. Only the addition of the engine file(s) and a small change to the initialization of the `VisualizationEngine` class would be needed.

Some example code of how the engine might work. It supposes a specific visualization engine for matplotlib and mayavi exist.

```
class visualization_engine():
```

```
def __init__(engine_type,*engine_args,**engine_kw_args):
    # First determine what engine we are using
    # and set the relevant functions to those defined in
    # another file such as nx_pylab.
    if engine_type == 'matplotlib':
        from nx.engines import nx_pylab
        self.engine = nx_pylab.start_engine(*engine_args,
                                            **engine_kwargs)
        ...
    elif engine_type == 'mayavi':
        from nx.engines import nx_mayavi
        self.engine ...
    ...
    self.node_color = {}
    self.node_size = {}
    ...

def parse_color(c):
    if type(c) is str:
        convert_to_tuple
    elif:
        ...
...

def draw_node(n,...):
    # handles color parsing and sends data to the engine.
    if n in not in node_color:
        node_color[n] = parse_color(...)
    ...
    engine.draw_node(n,node_color[n],...)
```

This generic class would be the interface used in most cases. No matter the drawing method used, the interface to it through networkx would be the same for the user. A user could then switch between matplotlib and mayavi by simply changing the `engine_type` variable in their code. This would also set up a standard template for new visualization engines to be written. Since all arguments are pre-parsed into a sensible format, a developer would simply have to write the appropriate functions for the particular engine platform.

2.3 Graph Interconnection

Graphs could then be initiated with an engine in mind, so subsequent modifications to the graph are immediately seen in the visualization.

```
G = nx.Graph(vis_engine='matplotlib') #Create Graph and start matplotlib engine
G.add_node(0) #Create node and draw in matplotlib engine
```

Alternatively, independent engines could be started, and graphs sent to them for visualization, with multiple graphs potentially being plotted using the same engine.

```
mpl_engine = nx.start_visualization_engine('matplotlib') #Start the Engine
mpl_engine.draw(G1) # Draw the graph G1 once
mpl_engine.draw_and_monitor(G2) #Draw the Graph G2 and any subsequent changes
G3=nx.Graph(vis_engine=mpl_engine) #Draw all changes to G3 on the mpl_engine
```

This would be especially useful in interpreter settings, and when testing out new graph creation models.

Internally, small changes to the NetworkX Graph class would be made. The class would have a new member which would be a visualization_engine. Calls which alter the state of the graph would then include a check the visualization engine, for instance:

```
def add_node(self, n, attr_dict=None, **attr):
    ...
       self.node[n].update(attr_dict) #Current end of add_node function
    if not self.visualization_engine is None:
        self.visualization_engine.draw_node(G)
```

2.4 A matplotlib Engine

In this framework a matplotlib engine would be created. While the current drawing methods seem to work adequately several features need to be added or improved include: appropriately drawn arrows, variable node size, edge labels, and interactive node positions.

2.4.1 Arrows

Currently, arrows are drawn as a slightly thicker portion of line at the end of a directed edge. Early attempts to alter the function included a tracking of nodes objects and their boundaries and the calculation of clipping paths, resulting in aesthetically pleasing arrows, but a much longer execution time. A precomputed determination of edge endpoints (based on node positions) would likely speed up this computation.

2.4.2 Variable Node Shape

NetworkX currently uses matplotlib's `scatter` function to draw nodes on screen. This allows only nodes of uniform shape to be drawn on the screen, and if multiple shapes are needed, repeated calls to `draw_nodes`, must be made. By having the matplotlib engine maintain nodes as patch objects their individual shape could be easily controlled

2.4.3 Edge labels

Edge labels, while included as an option in the draw documentation are currently not drawn rendered. This would simply require a determining appropriate positioning of edge labels, and including the code to render them.

2.4.4 Interactive Node Positions

Many visualization tools allow the user to select nodes and reposition them on the screen, or view any attributes that might be associated with a given node or edge. This project would investigate what would be required to implement a feature that would capture user input and act appropriately. While this is certainly a useful feature, it may be unrealistic to implement in the matplotlib library, due to computational time constraints. However in the current framework, changes to the matpotlib visualization engine would be all that is required.

2.5 Mentors

Here is a list of potential mentors and their GSoC mentor IDs:

- Aric Hagberg(ahagberg)
- Loïc Séguin-charbonneau (loicseguin)
- Dan Schult(dschult)
- Chris Ellison (hei7boht)

3 Benefits for NetworkX

NetworkX's intuitive API and large library of algorithms have made it very popular for investigation of complex networks. However, where it excels in graph analysis, it has fallen behind excellent visualization projects. By integrating with these other projects, and providing an easy interface to other visualization engines, NetworkX would be the easy choice for anyone doing research into complex networks.

4 Success Criteria

1. Creation of the classes described above, and modification to the current NetworkX classes.
2. Clear and concise documentation of all functions and changes.
3. The creation of other visualization engines for NetworkX
4. The eventual removal of old drawing functions from the NetworkX codebase.

5 Project Timeline

Pre-Coding Solicit advice from NetworkX mailing list and mentor about what drawing options should be available and sensible defaults, as well as any interface ideas

Week 1-2 Build `VisualizationEngine` class, and make modifications to NetworkX base classes to make use of visualization engine.

Week 3-4 Port current matplotlib drawing code into matplotlib visualization engine class.

Week 5-8 Improve matplotlib visualization engine so arrows are drawn appropriately, variable node shapes are possible, and edge labels are drawn. Solicit advice from the matplotlib mailing list about speed and drawing improvements.

Week 8-11 Testing, bug-fixes and documentation of the matplotlib visualization engine. Change current matplotlib drawing functions to their visualization engine equivalents. Investigate adding interactivity to the class.

Week 11-12 If time allows begin on other interfaces for visualization engine. This might include creating a PyGraphViz engine or exploring new engines such as mayavi, cytoscape, or gehpi.

6 Biography

6.1 Personal History

I received by bachelor's degree in Mathematics and Computer Engineering in 2006 from the South Dakota School of Mines and Technology. I am currently seeking a PhD in Computer Science from the University of New Mexico under Stephanie Forrest. My current research focuses on evaluating Internet growth using agent based models. I am also interested in the structure of social networks and how demographic processes create small world social networks, as well as graph models embedded in metric spaces.

6.2 Python

I use Python extensively in my research, as it allows for quick manipulation and analysis of data in a variety of formats. I have produced a number of useful libraries available called python_lib to provide additional functionality to matplotlib, parallel computation, and statistical computing. I have also collected several functions written by others (some of which I have modified) that have proven to be very useful.

6.3 NetworkX

I have used Python and NetworkX extensively in my research. I became involved in the NetworkX project in the Summer of 2010, and in the past year have made several contributions which are in the current codebase: Tickets #356, #323, #357, #375, and #388. I also have contributed code in several pending tickets and discussions (Tickets #378, #359, #345, #390, #387, #371, #396, #533, #360, #395, and #355). In particular I worked on Ticket #423 which attempts to address some of the problems with matplotlib drawing. Additionally, a mailing list discussion attempts to provide an openGL drawing method to NetworkX. This makes me very familiar with the codebase, and able to quickly develop new functionality for NetworkX.

6.4 Contact

Ben Edwards bedwards <at> cs <dot> unm <dot> edu Farris Engineering Center 355d University of New Mexico Albuquerque, NM 87131