Mission Control: An Open Source Usability Package for GraphSpace in Python

Nick Franzese
Reed College
Portland, Oregon
nick.franzese@gmail.com

ABSTRACT

GraphSpace is a highly customizable platform for graph visualization with a suite of helpful features. These features offer great potential for a variety of academic uses, but actualization of this potential is dampened by usability issues. Formatting data for visualization with GraphSpace can be daunting for those unfamiliar with coding, and can be a chore even for experienced programmers as each graph requires custom built code to visualize. For this reason I created Mission Control, an open source usability package for GraphSpace in Python. The package aims to significantly lower the usability barrier of GraphSpace while maintaining customizability. In the present paper I detail the user API of the Mission Control package and showcase a few graphs that I was able to visualize quickly and effortlessly through the package.

Keywords

Visualization; Python; GraphSpace; Open Source; Usability

1. MOTIVATION

Graphspace is a highly customizable platform for graph visualization with a suite of helpful features including cloud sharing with privacy options, manual and automatic layout control, a search feature for graph elements, graph tagging for organizational purposes, and perhaps most importantly a wide variety of visual graph attributes that can be controlled by the user programmatically [3]. These features offer great potential for scientists and mathematicians in search of a way to visualize graph data, but the actualization of this potential is dampened by usability issues. In its current state, GraphSpace utilizes a JSON parser in order to transform user specified visual attribute data into graphical form. This means that a user must manually configure code to set the visual attributes of each edge and node in the graph, configure a JSON file accordingly, and then upload the JSON file to the GraphSpace server. Presently there exist tools [1] to ease the processes of converting the visual attribute data to JSON and uploading it to the server, but setting the visual attributes themselves requires customized code for each new application. In order to set the visual attributes of the nodes (edges are similar), the user must compile a nested dictionary containing a key for each node whose value is a dictionary whose keys are GraphSpace visual attributes. The user must then set these GraphSpace visual attributes to appropriate values in order to reflect whatever trend in the data they are attempting to visualize. The advantage of this involved approach is that it is highly customizable, allowing users to visualize multidimensional data in varied and compelling ways. But this utility comes at a price. Formatting data for visualization with GraphSpace can be daunting for those unfamiliar with coding, and can be a chore even for experienced programmers.

For this reason I created Mission Control, an open source usability package for GraphSpace in Python. Acknowledging that customizability is a key factor in GraphSpace's attractiveness, I aimed to make Mission Control a convenient framework for controlling GraphSpace's visual attributes whilst maintaining the openness offered by its programmatic interface. To this end I implemented features such as a dynamic text file parser, a user-controlled system of default visual attributes, intelligent data read in, and an object class designed to dynamically handle the addition and removal of user-defined data attributes. As a result of these features, users of Mission Control can visualize multidimensional graph data in a variety of ways and upload it to GraphSpace with just a few operations.

2. METHODS

In order to establish a framework which can conveniently pattern visual attributes according to data whilst preserving some of the programmatic customizability offered by GraphSpace, I constructed an object class designed to handle the dynamic addition and removal of user-defined data attributes. I termed this class Generic Dynamic Object (GDO). The GDO keeps track of a directory and a data dictionary. The `newAttr()` GDO method adds a string to the directory. The `put()` GDO method takes an attribute name and a value, and adds that value to the data dictionary under the attribute name only if the attribute name already exists in the directory. The `get()` GDO method retrieves a value from the data dictionary by a given attribute name. This is the core framework which allows users to dynamically construct data attributes for the nodes and edges of the graph whilst maintaining a regular format which can be

Bio331 Fall 2016, Reed College, Portland, OR

ACM ISBN X-XXXXX-XX-X/XX/XX.
DOI: XX.XXX/XXX_X

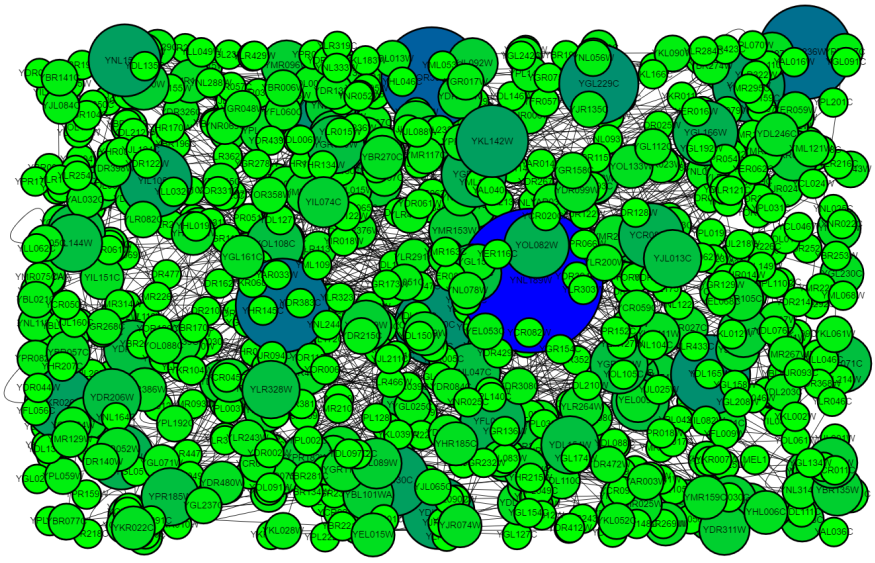

Figure 1: Uetz Screen Yeast Interactome visualized programmatically using Mission Control. Both size and background color (a gradient where green means low and blue means high) are patterned by node degree. This graph was visualized and uploaded using a total of 5 operations (parse, nodeInstall, visualize, visualize, upload) with the Mission Control package.

handled by the rest of the Mission Control system. Nodes and Edges are constructed as subclasses of the GDO which necessitate an ID and a source and target respectively upon initialization.

The Graph class is a wrapper which keeps track of the nodes and edges and contains a variety of utility functions. Data is compiled into a Graph containing Nodes and Edges by way of a dynamic text file parser function. The parser is designed to be able to handle both basic data read in from outside sources, and read in of exported data from working sessions of Mission Control containing user defined data attributes and visual attributes. To this end, the parser can read in a single text file containing only edges with user-specified delimiter and header (the header may be given as an argument in list form. It determines what columns of the text file are interpreted as which data attributes). It can also read in both an edge and a node file with multiple columns of data and headers describing the names of the attributes. The parser also supports a rudimentary typing system which intelligently determines whether numeric values in a data column should be floats or integers, and whether textual data should be boolean, None, or string type. The `parse()` function is the first component of the user API, as it must be used in order to construct the Graph. The rest of the user API is composed of methods of the Graph class, detailed as follows. For a summary of the user API, see Table 3.

Two Graph methods control dynamic data input from the user: `nodeInstall()` and `edgeInstall()`. The `nodeInstall()` method takes an attribute name and a dictionary whose keys are node IDs and whose values are data for the given attribute. It then creates a corresponding entry in each Node of the graph. This method supports intelligent data read in: if the supplied dictionary for a new user-defined data attribute does not contain all of the nodes in the graph, the excluded nodes will gain the attribute with value `None`. If the data attribute has already been defined and the dictionary does not contain all the nodes in the graph, the included nodes will be updated with the given values and the excluded nodes will be left alone. Data attributes with value `None` are handled gracefully by the system of default visual attributes as will be discussed below. The `edgeInstall()` method works similarly to the `nodeInstall()` method.

The `visualize()` Graph method takes the name of a data attribute and leads the user through a rudimentary user interface to determine whether the given data is continuous or discrete, and which visual attribute that the user would like to employ to represent the data. Continuous node data can be visualized by GraphSpace visual attributes attributes `background_color, border_color` (according to a gradient constructed out of two user-supplied RGB color vectors), `background_blacken`, and `size`. Discrete node data can be visualized by GraphSpace visual attributes `background_color`, `border_color` (colors for discrete groups can be picked manually via user input or automatically via a color picking function), and `shape`. Continuous edge data can be visualized by `line_color` and `width`, and discrete edge data can be visualized by `line_color` and `line_style`. After walking through the user interface, the Graph records values for visualization as specially marked data attributes. These marked attributes are picked up by a few methods which convert these data into a dictionary formatted for JSON conversion during graph upload.

The `upload()` Graph method takes no arguments and leads the user through a rudimentary user interface in order to acquire GraphSpace username and password, graph title, graph ID, description, and tags. Once the user has supplied this information, the `upload()` method references the data attributes marked as important for visualization and constructs dictionaries formatted for JSON conversion accordingly as discussed above. The method then calls func-

Table 1: Toy Example Node Data

Node ID	Team	Node Degree	Random Float [0,50]
a	Alice	3	16.7
b	Bob	4	9.1
c	Bob	3	41.4
d	Alice	3	12.6
e	Alice	3	30.6
f	Bob	1	3.6
g	Alice	1	1.29
h	None	1	24.5
i	None	1	1.8
j	Bob	0	0.3

Table 2: Toy Example Edge Data

source	target	weight
b	a	10
d	a	5
c	a	2.5
d	c	1
c	b	1
d	b	3
e	b	1
g	e	7
f	e	1
i	h	4

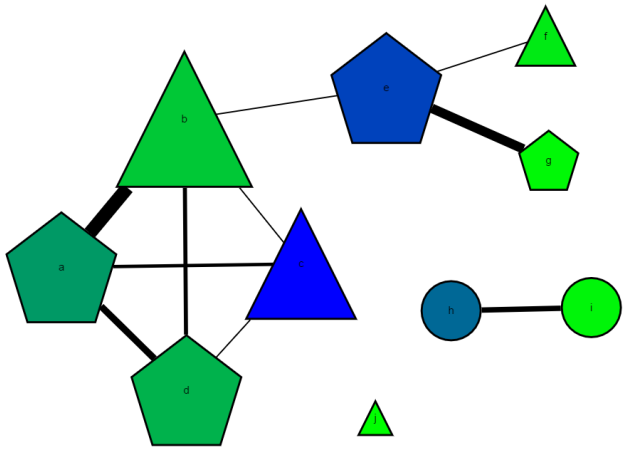

Figure 2: Visual representation of a multidimensional toy example graph. Team is patterned as node shape (Alice = pentagon, Bob = triangle, default = circle), Random Float [0,50] is patterned as background color (gradient, blue means high and green means low), Node Degree is patterned as node size, and edge weight is patterned as edge width.

tions previously created for JSON formatting and upload [1] to push the graph to the GraphSpace server.

The `export()` Graph method takes two file names and constructs two text files encoding the nodes and edges with all of the currently saved data attributes. These files can be read by the `parse()` function to replicate the data attributes of a given Mission Control session. This is done through a line 1 header which is detected by the parser automatically.

The `default()` Graph method takes the name of a GraphSpace visual attribute and a value. It then updates a dictionary of default visual attribute values accordingly. These default values are referenced not only when a visual attribute has not been set, but also when a visual attribute has been set but a particular node or edge has None for that value. This means that a user can easily visualize discrete groups of nodes (e.g. cliques) without having to worry about exhaustively covering all of the nodes.

The `display()` Graph method can take no arguments, or a key string (`data, visual, default, nodes,` or `edges`) and prints a generalized or specific summary of Graph attributes accordingly.

The `remove()` Graph method takes the name of an attribute and removes it from the directory and data dictionaries of the nodes or edges as appropriate. This method is useful if a user only wants to `export()` a subset of the visual or data attributes from a given Mission Control session.

The `nodeGet()` and `edgeGet()` methods take the name of a data attribute and return the dictionary of values associated with that data attribute.

3. RESULTS & DISCUSSION

To demonstrate some of the utility of the Mission Control package, I have visualized some real biological data as well as some toy example data. The real data is from the Harvard Yeast Interactome Database and was gathered using a Uetz screen [2]. The data set contains 2,358 interactions among 1,548 proteins. Using Mission Control, I visualized this interactome, patterning both node size and node background color according to node degree (Figure 1). Going from the raw Yeast Interactome Database text file to this graphic was a simple matter of writing a function to find node degree, compiling the node degree values into a dictionary, and then calling a grand total of 5 operations from the Mission Control user API (`parse, nodeInstall, visualize, visualize, upload`). Without the Mission Control package, visualization would have required custom built code to parse the text file and then compile a set of GraphSpace attribute dictionaries which could then be processed into JSON format. The gain in convenience by using Mission Control is palpable.

Since I did not do any interesting analysis on the interactome data, Figure 1 does not fully characterize the capabilities of the Mission Control package to represent multidimensional data. For this reason, I present a toy example graph with a few additional data attributes (Tables 1 & 2). Figure 2 patterns Team as node shape, Random Float [0,50] as background color according to a gradient from green to blue, Node Degree as node size, and weight as edge width, thus representing several attributes of data in one succinct graphic. Note that nodes h and i have None as their Team value, and thus they are set to the default node shape (ellipse). Thanks to the functionality granted by the dynamic parser, the additional data attributes (Team, Node Degree, Random Float [0,50], and weight) could be input through the parsed text files. Thus Figure 2 took only 6 Mission Control operations (`parse, visualize, visualize, visualize, visualize, upload`) to compose and upload to GraphSpace.

4. CONCLUSION

The Mission Control package offers a great step forward in terms of the usability of GraphSpace. Its construction imparts significant convenience for users attempting to visualize graph data while maintaining a high degree of customizability. The package compiles a solid foundation of core features, and is eminently useful in its present state. Future directions for development may also include layout construction features (patterned according to data attributes), and advanced user tools to make the package fully and intuitively compatible with manual programmatic customizability.

5. REFERENCES

[1] A Ritz. Bio 331 utils. https://github.com/annaritz/bio331-utils, 2016.

[2] B. Schwikowski, P. Uetz, and S. Fields. A network of protein-protein interactions in yeast. *Nat. Biotechnol.*, 18(12):1257–1261, Dec 2000.

[3] D Singh, A Ritz, A Tegge, C Poirel, P Kraikivski, N Adames, K Luther, S Kale, J Peccoud, J Tyson, and TM Murali. Graphspace: Collaborating through networks on the web. *Unpublished Work*.

Table 3: Summary of the Mission Control User API

Function Name	Important Arguments	Description	Graph Method?
parse()	edgefile, nodefile, delimiter	Returns properly formatted text file data as a Graph object	No
nodeInstall()	attrName, valueDict	Assigns each value in valueDict to data attribute attrName for nodes	Yes
edgeInstall()	attrName, valueDict	Assigns each value in valueDict to data attribute attrName for edges	Yes
visualize()	attrName	Patterns a visual attribute after the data attribute given by attrName	Yes
upload()	No Arguments	Uploads the currently visualized graph to GraphSpace	Yes
export()	edgefile, nodefile	Saves all data attributes from the current session in two files with the given names	Yes
default()	GS_attr, value	Sets the default value for the given visual attribute GS_attr to the given value	Yes
display()	c	Gives a summary of the Graph with different options depending on the control string c	Yes
remove()	attrName	Removes the data attribute given by attrName from the Graph	Yes
nodeGet()	attrName	Returns the node data attribute given by attrName in dictionary form	Yes
edgeGet()	attrName	Returns the edge data attribute given by attrName in dictionary form	Yes