Chapter 1

NetworkX Graph Library

Contributed by Satyaki Sikdar

1.1 Background

NetworkX is an open-source Python library designed to handle and explore graphs [1]. The project started in 2005 at the Los Alamos National Laboratory by Aric Hagberg and Pieter J. Swart. It is still in active development with frequent releases. It is available for download from their official website `www.networkx.github.io`.

The philosophy behind the design of NetworkX was to create a tool which is user-friendly, therefore making it accessible to a large community of researchers in complex networks which included physicists, social scientists, mathematicians, and computer scientists. However, it does suffer from scalability problems, both regarding computation time and memory, when it comes to handling large million node graphs for example. However, it can handle small to medium scale graphs with ease and has a vibrant library of built-in graph algorithms and generators.

1.2 Expressing Graphs

Out of the box, NetworkX supports 4 primary graph containers—`Graph, DiGraph, MultiGraph` and `MutiDiGraph` representing undirected, directed, multi, and multi-directed graphs respectively. The graphs are represented using nested Python `dictionary` objects. This allows users to add an arbitrary number of node and attributes which becomes very useful while handling heterogeneous networks. Self-loops are not allowed only in the `Graph` class.

Any hashable object can act as a node in the graphs. Thus it does not require the graphs to have continuous numeric node labels like the usual C/C++ based graph libraries do. Edges can be created between non-existing nodes in the graph, which then get created. It supports a lot of different file formats for reading and writing graphs including the conventional edge lists, adjacency lists, GEXF, GML, JSON, Pajek, GraphML among others[1]. This makes NetworkX very useful since there exist different graph representation formats, with each having their pros and cons.

[1]find the full list here `https://networkx.github.io/documentation/stable/reference/readwrite/index.html`

1.3 Syntax

Since NetworkX is an external library, one must import it to the current namespace before using it by using the command `'import networkx as nx'` (`nx` is a popular nickname of the library). Table 1.1 lists some of the common NetworkX library methods.

1.4 Key Graph Primitives

Discuss here what are the key graph primitives supported by the paradigm.

1.5 Execution Model

As mentioned before, graphs are stored as nested dictionary objects. Since the whole source code is in pure Python, it relies heavily on Python functions and data structures. For example, the graph primitives are defined as regular Python classes. It does, however, use functions from the Numpy and SciPy libraries to perform specific tasks like eigen-decompositions and other linear algebra operations. So, by design, NetworkX follows a serial computation model. This allows the API to be simple and user-friendly while sacrificing performance for large graphs.

1.6 Examples

The following code shows the implementation of the Breadth-First Search routine on a NetworkX graph: `G`, which is a Barabasi-Albert graph with 100 nodes. This is one of many built-in graph generators; see the full list here[2].

NetworkX code for a Breadth First Search routine

```
import networkx as nx
from collections import deque

def BFS(G, s):
    """
    Runs BFS from source node 's'.
    Returns the shortest path dictionary 'd'
    """
    d = {} # stores the shortest distance from source 's'
    d[s] = 0

    Q = deque() # creates a new Queue object
    Q.append(s) # adds the source 's' to the queue

    while len(Q) != 0:
        u = Q.popleft() # dequeue from queue
        for v in G.neighbors(u): # iterate thru the neighbors of 'u'
            if v not in d: # 'v' is unvisited
                d[v] = d[u] + 1 # update shortest path distance of 'v'
                Q.append(v) # add 'v' to the queue
    return d
```

[2]`https://networkx.github.io/documentation/stable/reference/generators.html`

```
G = nx.barabasi_albert_graph(100, 3) # creates a BA graph with n=100, m0=3
d = BFS(G, 0) # run BFS from node 0
print(d) # prints the dictionary
```

Developers of the *graph-tool* graph library have benchmarked the performance of NetworkX and other graph libraries in Python by comparing the running times of graph algorithms like PageRank, single source shortest path, K-core decomposition, etc[3]. NetworkX is the only library that is written in pure Python, so it is considerably slower than the rest.

1.7 Conclusion

NetworkX provides a flexibly and easy to use framework for handing small to medium scale graphs. The API is mature and has an extensive list of in-built graph algorithms and generators.

[3]more details here `https://graph-tool.skewed.de/performance`

Table 1.1: Some common NetworkX function calls

	Description
Graph containers	
`G = nx.Graph()`	`G` is an empty undirected graph
`G = nx.DiGraph()`	`G` is an empty directed graph
`G = nx.MultiGraph()`	`G` is an empty undirected multi-graph
`G = nx.MultiDiGraph()`	`G` is an empty directed multi-graph
Nodes	
`G.add_node('a')`	adds a new node with id `a`
`G.add_nodes_from(iterable`	adds nodes from the iterable to the graph
`G.add_nodes_from(H)`	adds nodes from another NetworkX graph *H*
`G.nodes()`	returns an iterable *view* of the nodes in the graph
`G.remove_node(id)`	removes an *existing* node `a` and all its edges from the graph
`G.remove_nodes_from(iterable)`	removes existing nodes with the ids taken from the iterable
`G.has_node(id)`	checks if the node with `id` exists in the graph
Edges	
`G.add_edge(u, v)`	adds the edge `(a, b)` to the graph. Replaces existing edge if it already exists.
`G.add_edges_from(iterable)`	adds edges from the iterable with items of the form `(u, v)` or `(u, v, w)`
`G.add_edges_from(H)`	adds edges from another NetworkX graph *H*
`G.edges()`	Returns an iterable *view* of the edges in the graph
`G.remove_edge(u, v)`	removes an *existing* edge `(u, v)` from the graph
`G.remove_nodes_from(iterable)`	removes existing edges from the iterable
`G.has_edge(u, v)`	checks if the edge `(u, v)` exists in the graph
Neighbors	
`G.neighbors(u)`	returns an iterable *view* of the neighboring nodes of `u`
`G.predecessors(u)`	returns an iterable *view* of the predecessor nodes of `u` in a directed graph
`G.successors(u)`	returns an iterable *view* of the successors nodes of `u` in a directed graph
Graph Properties	
`G.order()`	returns the number of nodes in the graph
`G.size()`	returns the number of edges in the graph
`nx.diameter(G)`	returns the diameter of the graph

Bibliography

[1] Aric Hagberg, Pieter Swart, and Daniel S Chult. Exploring network structure, dynamics, and function using networkx. Technical report, Los Alamos National Lab.(LANL), Los Alamos, NM (United States), 2008.